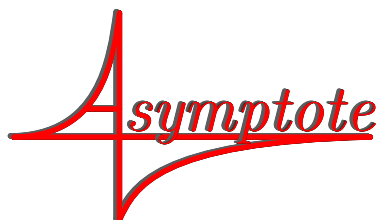


# Asymptote: 矢量绘图语言

---

1.91 版



本文件是 Asymptote 1.91 版的文档。

<http://asymptote.sourceforge.net>

版权所有 © 2004–9 Andy Hammerlindl, John Bowman, and Tom Prince.

在 GNU Lesser General Public License (GNU 较宽松公共许可证) 的条款下 (见源文件目录顶层的 LICENSE 文件) 授予复制、发布和/或修改此文档的许可。

翻译: 刘海洋 (leoliu.pku@gmail.com)      2010 年 1 月 1 日

## 翻译说明

本手册根据 Asymptote 1.91 手册节选翻译。目前节选的部分是原手册的三、四、五、六章，包括全部的基本绘图功能说明，但未包括软件配置、与  $\text{\LaTeX}$  结合、扩展模块、命令行等相关内容。

手册大部分取原文直译，在不影响意义时偶有变化，或增添少量词语帮助理解。个别术语夹注原文。增加语法高亮，大部分格式如旧，格式上偶有不妥处迳改，不另注出。原文明显有误的，更改后用译注说明。

水平所限，本译文错漏难免，有疑义请以原文为准。

# 目录

<b>第三章 教程</b>	<b>1</b>
3.1 在批处理模式中绘图 . . . . .	1
3.2 在交互模式中绘图 . . . . .	1
3.3 图形尺寸 . . . . .	2
3.4 标签 . . . . .	3
3.5 路径 . . . . .	4
<b>第四章 绘图命令</b>	<b>7</b>
4.1 描绘 (draw) . . . . .	7
4.2 填充 (fill) . . . . .	9
4.3 剪裁 (clip) . . . . .	12
4.4 标注 (label) . . . . .	12
<b>第五章 Bézier 曲线</b>	<b>17</b>
<b>第六章 编程</b>	<b>19</b>
6.1 数据类型 . . . . .	20
6.2 路径 (path) 与路向 (guide) . . . . .	27
6.3 画笔 . . . . .	36
6.4 变换 . . . . .	46
6.5 帧 (frame) 与图 (picture) . . . . .	47
6.6 文件 . . . . .	54
6.7 变量初始式 . . . . .	57
6.8 结构体 . . . . .	58
6.9 运算符 . . . . .	62
6.9.1 算术和逻辑运算符 . . . . .	62
6.9.2 自 (赋值) 与前缀运算符 . . . . .	63
6.9.3 用户自定义运算符 . . . . .	64
6.10 隐式放缩 . . . . .	64
6.11 函数 . . . . .	65

6.11.1 默认参数 . . . . .	67
6.11.2 具名参数 . . . . .	68
6.11.3 剩余参数 . . . . .	69
6.11.4 数学函数 . . . . .	70
6.12 数组 . . . . .	72
6.12.1 切割 . . . . .	79
6.13 类型转换 . . . . .	80
6.14 导入 . . . . .	82
6.15 静态 (static) . . . . .	84
<b>索引</b>	<b>86</b>

## 第三章 教程

### 3.1 在批处理模式中绘图

要绘制一条从坐标 (0,0) 到坐标 (100,100) 的直线，建立一个名为 test.asy 的文本文档，它包含：

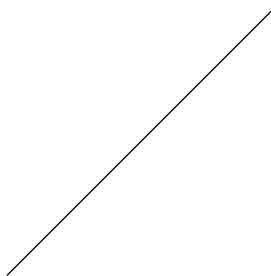
```
draw((0,0)--(100,100));
```

然后执行命令

```
asy -V test
```

此外，Windows 用户<sup>1</sup> 可以将 test.asy 拖放到桌面的 asy 图标上（或是把 Asymptote 作为扩展名为 asy 的文件的默认应用程序）。

这种方法，即批处理模式，输出一个 PostScript 文件 test.eps。-V 选项打开一个 PostScript 查看器窗口，从而你可以立即看到结果：



-- 连接符把两个点 (0,0) 和 (100,100) 用一条线段连接。

### 3.2 在交互模式中绘图

另一种途径是交互模式，Asymptote 会一条条读入用户敲入的命令。要试试这个，可以点击 Asymptote 图标或输入命令 asy 进入 Asymptote 的交互模式。然后输入

```
draw((0,0)--(100,100));
```

并按回车键，就得到上面的图形。这里你可以进一步输入 draw 命令，它们将被加到上面展示出的图中，输入 erase 清空画面，输入

---

<sup>1</sup>原手册误作 MSDOS，此更正。——译者注

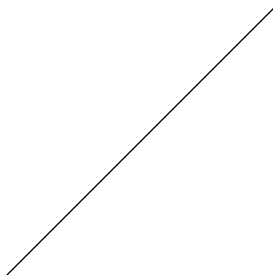
```
input test;
```

会运行文件 `test.asy` 中包含的所有命令，或是输入 `quit` 退出交互模式。你可以在交互模式中使用方向键来编辑以前的行。Tab 键将自动补全无歧义的词；否则，再次按下 tab 将显示可能的选择。交互模式进一步的特有命令在原手册第 10 章 Interactive Mode<sup>2</sup> 描述。

### 3.3 图形尺寸

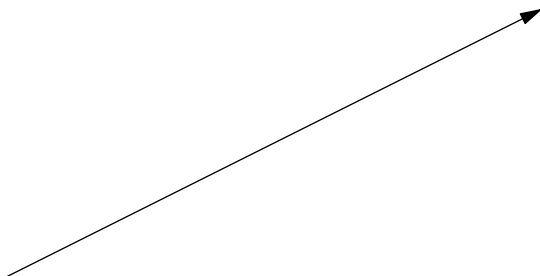
在 Asymptote 中，像  $(0,0)$  和  $(100,100)$  这样的坐标称为复数<sup>3</sup>，以 PostScript “大点”<sup>4</sup> ( $1\text{ bp} = 1/72\text{ inch}$ ) 为单位，且默认的线宽为  $0.5\text{ bp}$ 。然而，直接以 PostScript 坐标通过是不方便的。下面的例子通过放缩直线  $(0,0) \rightarrow (1,1)$ ，将其匹配到一个宽  $100.5\text{ bp}$  高  $100.5\text{ bp}$  的矩形大小（多出的  $0.5\text{ bp}$  被直线宽占用），来生成与前面例子完全相同的输出：

```
size(100.5,100.5);
draw((0,0)--(1,1));
```



可以按 `pt` ( $1\text{ pt} = 1/72.27\text{ inch}$ )、`cm`、`mm` 或是 `inches`<sup>5</sup> 来设定尺寸。两个非零尺寸参数（或单个尺寸参数）限定两个方向的尺寸，并保持宽高比例。如果给定尺寸（`size`）参数为 0，则该方向没有约束；总的放缩将由另一方向决定（见 47 页 6.5 节 `size` 函数说明）：

```
size(0,100.5);
draw((0,0)--(2,1),Arrow);
```



<sup>2</sup>本译本未包括此章。——译者注

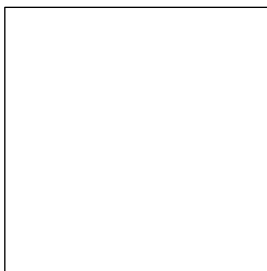
<sup>3</sup>原文为 `pair`，即序对。译文一般通称为复数。——译者注

<sup>4</sup>这里的单位大点（big point, `bp`），即  $1/72\text{ inch}$ ，其概念来自  $\text{T}_{\text{E}}\text{X}$ 。在  $\text{T}_{\text{E}}\text{X}$  之外的印刷领域，大点就叫点（point, `pt`），而在  $\text{T}_{\text{E}}\text{X}$  中的点要小一些： $1\text{ pt} = 1/72.27\text{ inch}$ 。——译者注

<sup>5</sup>这里 `inch` 和 `inches` 意义相同，都是英寸。——译者注

要把若干个连接起来创建一条环形路径，使用 `cycle` 关键字：

```
size(100,100);
draw((0,0)--(1,0)--(1,1)--(0,1)--cycle);
```



为方便计，路径  $(0,0)--(1,0)--(1,1)--(0,1)--cycle$  可以用预定义的变量 `unitsquare` 代换，或等价地，`box((0,0),(1,1))`。

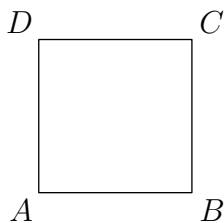
要使用户坐标正好表示 `1cm` 的倍数：

```
unitsize(1cm);
draw(unitsquare);
```

## 3.4 标签

在 Asymptote 中很容易增加标签；用双引号引起的 L<sup>A</sup>T<sub>E</sub>X 字符串、坐标和可选的对齐方向来设定标签：

```
size(3cm);
draw(unitsquare);
label("$A$", (0,0), SW);
label("$B$", (1,0), SE);
label("$C$", (1,1), NE);
label("$D$", (0,1), NW);
```

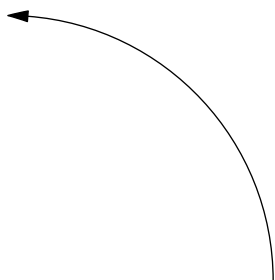


Asymptote 使用标准的罗盘方向  $E=(1,0)$ 、 $N=(0,1)$ 、 $NE=unit(N+E)$  以及  $ENE=unit(E+NE)$  等等，并在 Asymptote 基本模块 `plain` 中定义有复数 (`pair`) `up`、`down`、`right` 和 `left` (具有名为 `E` 的局部变量的用户可以通过在前面缀以定义它的模块名来访问罗盘方向 `E: plain.E`)。

### 3.5 路径

这个例子绘制一条路径近似为  $1/4$  圆，末端有箭头：

```
size(100,0);
draw((1,0){up}..{left}(0,1),Arrow);
```



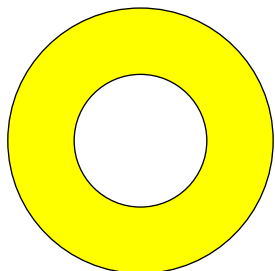
这里花括号中的方向 `up` 和 `left` 分别指定了在点  $(1,0)$  和  $(0,1)$  进入和发出的方向。

一般地，一条路径由一系列复数（或其他路径）用 `--` 相连来表示直线段，或用 `..` 相连表示三次样条（参见 17 页第 五 章）。设定最终的 `..cycle` 建立一条平滑连回初始结点的环形路径，如在这里对单位圆的近似（精确到 0.06% 以内）：

```
path unitcircle=E..N..W..S..cycle;
```

一条相连的 Asymptote 路径，与一条 PostScript 的 `subpath`（子路径）等价。`^^` 二元运算符，它要求画笔从左边路径的终点移动（而不绘制出来或影响终点的曲率）到右边路径的起点，可以用来将若干 Asymptote 路径汇集到一个 `path[]` 数组中（等价于一条 PostScript 路径）：

```
size(0,100);
path unitcircle=E..N..W..S..cycle;
path g=scale(2)*unitcircle;
filldraw(unitcircle^^g,evenodd+yellow,black);
```



这里 PostScript 奇偶（`evenodd`）填充规制设定仅在两个圆<sup>6</sup>之间界定的区域会被填充。在这个例子中，如果小心地变换路径的方向，可以使用默认的零卷绕数（`zerowinding`）填充规则得到同样的效果：

<sup>6</sup>原文作两个单位圆，但事实上这里其中一个圆已经放缩过。——译者注



```
filldraw(unitcircle^^reverse(g),yellow,black);
```

`^^` 运算符被模块 `three.asy` 中的 `box(triple, triple)` 函数用于无回溯步骤地构造立方体 `unitbox` 的边（见原手册 8.29 节 `three` 模块<sup>7</sup>）：

```
import three;
dotgranularity=0; // Render dots as spheres.

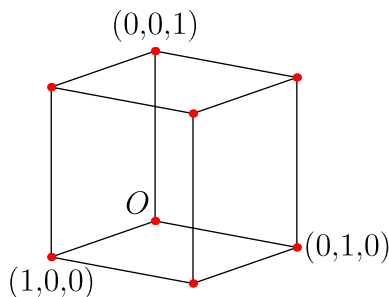
currentprojection=orthographic(5,4,2,center=true);

size(5cm);
size3(3cm,5cm,8cm);

draw(unitbox);

dot(unitbox,red);

label("$O$", (0,0,0),NW);
label("(1,0,0)", (1,0,0),S);
label("(0,1,0)", (0,1,0),E);
label("(0,0,1)", (0,0,1),Z);
```



更多的例子见原手册 8.27 节 `graph` 模块<sup>8</sup>（或在线的 Asymptote 画廊和在 <http://asymptote.sourceforge.net> 发布的外部链接），包括二维和三维的科学图形。额外的例子已由 Philippe Ivaldi 在 <http://www.piprime.fr/asymptote> 发布。一份由用户编写的 Asymptote 绝佳教程又可得自<sup>9</sup>：[http://www.artofproblemsolving.com/Wiki/index.php/Asymptote:\\_Basics](http://www.artofproblemsolving.com/Wiki/index.php/Asymptote:_Basics)

<sup>7</sup>本译本未包括此节。——译者注

<sup>8</sup>本译本未包括此节。——译者注

<sup>9</sup>这个教程是比较简单的。在中文社区，由用户编写的 Asymptote 教程包括较早莫图（lapackapi@ctex.org）的一个中途已中止的项目《Asymptote 作图指南》；还有 cvgmt@ctex.org 与译者目前正在编写的两个教程，连同本文的最新译本则可在 asy4cn 项目找到。中文译介的资料包括本手册和由 goodluck@ctex.org 翻译的 Asymptote 官方 FAQ。这些中文资料一般都可以在 CTeX 论坛找到作者并获得更新和支持。——译者注



## 第四章 绘图命令

Asymptote 的全部绘图功能都基于四个基本命令。三个 PostScript 绘图命令 `draw`、`fill` 和 `clip` 按执行顺序将对象加入图，最近绘制的对象出现在顶层。标注命令 `label` 可用来添加文字标签和外部 EPS 图像，它会显示在 PostScript 对象的上面（鉴于这通常是人们想要的），而又会以它们执行的相对次序显示。在一幅图（`picture`）上绘制对象后，图可以由 `shipout` 函数输出（见 47 页 6.5 节 `shipout` 的说明）。

如果需要在标签（或逐字的 T<sub>E</sub>X 命令，见 47 页 6.5 节 `tex` 的说明）上面绘制 PostScript 对象，可以使用 `layer` 命令开始一个新的 PostScript/L<sup>A</sup>T<sub>E</sub>X 层：

```
void layer(picture pic=currentpicture);
```

`layer` 函数给人对象绘制的完全控制。层（`layer`）依次绘制，最近的层出现在顶层。在每个层中，标签、图像、逐字 T<sub>E</sub>X 命令总在该层中 PostScript 对象之后绘制。

尽管下面一些绘图命令有很多选项，它们都有合理的默认值（例如，图参数默认为 `currentpicture`）。

### 4.1 描绘（`draw`）

```
void draw(picture pic=currentpicture, Label L="", path g,
          align align=NoAlign, pen p=currentpen,
          arrowbar arrow=None, arrowbar bar=None, margin margin=NoMargin,
          Label legend="", marker marker=nomarker);
```

在图 `pic` 上使用画笔 `p` 描绘路径 `g`，并带可选的绘制属性（标签 `L`，显式标签对齐 `align`，箭头与短杠 `arrow` 和 `bar`，边距 `margin`，图例（`legend`），以及记号 `marker`）。只有一个参数，即路径，是必需的。为方便计，参数 `arrow` 与 `bar` 可按任何次序限定<sup>1</sup>。参数 `legend` 是构造可选图例项时使用的标签。

短杠在标明尺寸时很有用。`bar` 可取值 `None`、`BeginBar`、`EndBar`（或等价地 `Bar`）以及 `Bars`（在路径两端绘制短杠）。每个短杠设定（除了 `None`）将接受一个可选的实数参数以 PostScript 坐标标明短杠的长度。设置的短杠长度为 `barsize(pen)`。

`arrow` 可取值 `None`、`Blank`（不画出箭头与路径<sup>2</sup>）、`BeginArrow`、`MidArrow`、`EndArrow`（或等价地 `Arrow`）以及 `Arrows`（在路径的两端都绘制箭头）。除 `None` 和 `Blank` 外的所有箭头设定可以给以

<sup>1</sup>即，不论使用的是 `draw(g, BeginBar, EndArrow)` 还是 `draw(g, EndArrow, BeginBar)`，得到的图形效果是一样的。——译者注

<sup>2</sup>利用它可以只画一组短杠。——译者注

可选的箭尖参数 `arrowhead` (预定义的箭头风格 `Default`、`SimpleHead`、`HookHead`、`TeXHead` 之一), 实数 `size` (PostScript 坐标的箭尖尺寸), 实数 `angle` (箭尖角度, 以度计), 填充类型 `filltype` (`FillDraw`、`Fill`、`NoFill`、`UnFill`、`Draw` 之一) 以及 (除了 `MidArrow` 和 `Arrows`) 一个表示箭头顶端沿路径放置位置的实数 `position` (在 `point(path p, real t)` 意义下)。用画笔 `p` 绘制时, 默认的箭尖尺寸为 `arrowsize(p)`。还有 `size` 与 `angle` 默认值略有修改的适合曲线的箭头版本<sup>3</sup>: `BeginArcArrow`、`EndArcArrow` (或等价地 `ArcArrow`)、`MidArcArrow` 以及 `ArcArrows`。

边距可以用于收缩 `labelmargin(p)` 长的路径可见部分以避免与其他画出的对象重叠。`margin` 的典型值为 `NoMargin`、`BeginMargin`、`EndMargin` (或等价地 `Margin`) 以及 `Margins` (它在路径的两端都留下边距)。可以使用 `Margin(real begin, real end)`, 以用于对齐标签的单位 `labelmargin(p)` 的倍数来分别设定起始和终点的边距大小。此外, `BeginPenMargin`、`EndPenMargin` (或等价地 `PenMargin`)、`PenMargins`、`PenMargin(real begin, real end)` 以画笔线宽为单位确定边距, 这里考虑在绘制路径或箭头时的画笔线宽。例如, 使用 `DotMargin`, 即 `PenMargin(-0.5*dotfactor, 0.5*dotfactor)` 的缩写, 来绘制从通常的起点恰好到达宽度为 `dotfactor*linewidth(p)` 的终点的边界。修饰符 `BeginDotMargin`、`EndDotMargin` 和 `DotMargins` 与之类似。修饰符 `TrueMargin(real begin, real end)` 允许人们直接用 PostScript 单位设定边距, 而与画笔的线宽无关。

箭头、短杠和边距的用法在示例文件 `Pythagoras.asy`、`sqrtx01.asy` 和 `triads.asy` 中展示<sup>4</sup>。

一幅图 `pic` 的图例可以用下列过程对齐装入一帧:

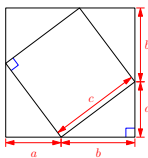
```
frame legend(picture pic=currentpicture, int perline=1,
    real xmargin=legendmargin, real ymargin=xmargin,
    real linelength=legendlinelength,
    real hskip=legendhskip, real vskip=legendvskip,
    real maxwidth=0, real maxheight=0,
    bool hstretch=false, bool vstretch=false, pen p=currentpen);
```

这里 `xmargin` 与 `ymargin` 设定四周  $x$  与  $y$  边距, `perline` 设定每行的条目数 (默认为 1; 0 表示自动选取此数目), `linelength` 设定路径线的长度, `hskip` 与 `vskip` 设定行距 (按图例条目尺寸的倍数), `maxwidth` 与 `maxheight` 设定可选的最终图例长宽上界 (0 表示无限定), `hstretch` 和 `vstretch` 允许图例在水平和垂直方向伸缩, 而 `p` 设定用于绘制边框的画笔。图例的帧于是可以使用 `add` 和 `attach` 添加进来并相对图 `dest` 上一点对齐 (参看 47 页 6.5 节 `add` 函数)。

要绘制一点, 只要画出只包含一个单独点的路径。在 `plain` 模块中定义的命令 `dot` 画出一个点, 它有等于显式画笔线宽或默认线宽放大 `dotfactor` (默认为 6) 倍的直径, 并使用设定的填充类型 (见

<sup>3</sup>其中如果使用 `TeXHead`, 尺寸过大, 而效果不佳。——译者注

<sup>4</sup>这里仅给出 `Pythagoras.asy` 例子的效果, 代码略:



47 页 6.5 节填充类型的说明):

```
void dot(picture pic=currentpicture, pair z, pen p=currentpen,
         filltype filltype=Fill);
void dot(picture pic=currentpicture, Label L, pair z, align align=NoAlign,
         string format=defaultformat, pen p=currentpen, filltype filltype=Fill);
void dot(picture pic=currentpicture, Label[] L=new Label[], pair[] z,
         align align=NoAlign, string format=defaultformat, pen p=currentpen,
         filltype filltype=Fill)
void dot(picture pic=currentpicture, Label L, pen p=currentpen,
         filltype filltype=Fill);
```

如果给定变量 `Label` 为第二个过程的 `Label` 参数, `format` 参数将用于在格式化打点位置的字符串 (这里默认值 `defaultformat` 为 `"$%.4g$"`)。第三个过程在复数数组 `z` 的每个点处画一个点。还可以在一条路径的每个结点处打一个点:

```
void dot(picture pic=currentpicture, Label[] L=new Label[],
         path g, align align=RightSide, string format=defaultformat,
         pen p=currentpen, filltype filltype=Fill);
```

参看原手册 7.25 节 `graph` 模块和 7.8 节 `markers` 模块<sup>5</sup>中给路径结点标记的更一般的途径。

要关于用户坐标 `origin` 绘制固定尺寸的对象 (以 PostScript 坐标), 使用过程

```
void draw(pair origin, picture pic=currentpicture, Label L="", path g,
          align align=NoAlign, pen p=currentpen, arrowbar arrow=None,
          arrowbar bar=None, margin margin=NoMargin, Label legend="",
          marker marker=nomarker);
```

## 4.2 填充 (fill)

```
void fill(picture pic=currentpicture, path g, pen p=currentpen);
```

使用画笔 `p`, 在图 `pic` 上填充由环形路径 `g` 限定的区域内部。

还有一个方便的 `filldraw` 命令, 它填充路径并绘制边界线。可以为每个操作分别设定画笔:

```
void filldraw(picture pic=currentpicture, path g, pen fillpen=currentpen,
              pen drawpen=currentpen);
```

这个固定尺寸的 `fill` 版本允许人们填充一个关于用户坐标 `origin`, 以 PostScript 坐标描述的对象:

```
void fill(pair origin, picture pic=currentpicture, path g, pen p=currentpen);
```

这只不过是如下命令的方便缩写:

---

<sup>5</sup>本译本未包括此部分。——译者注

```

picture opic;
fill(opic,g,p);
add(pic,opic,origin);

```

过程

```

void filloutside(picture pic=currentpicture, path g, pen p=currentpen);

```

填充路径  $g$  的区域外部，向外直达图  $pic$  的当前边界。

在二维画笔数组  $p$  之上使用填充规则 `fillrule` 的格状梯度光滑渐变，可以由

```

void latticeshade(picture pic=currentpicture, path g, bool stroke=false,
    pen fillrule=currentpen, pen[] p);

```

产生。如果 `stroke=true`，填充区域就与由 `draw(pic,g,fillrule+zerowinding)` 绘制的区域相同<sup>6</sup>；在这种情况下，路径  $g$  不必是环形的。 $p$  中的画笔必须属于同一颜色空间。可以使用函数 `rgb(pen)` 或 `cmyk(pen)` 来将画笔提升到一个较高的颜色空间，如在示例文件 `latticeshading.asy` 中展示的那样<sup>7</sup>。

按线段  $a$ -- $b$  方向，从画笔  $pen_a$  到  $pen_b$  的轴向梯度光滑渐变可以由下面函数得到<sup>8</sup>：

```

void axialshade(picture pic=currentpicture, path g, bool stroke=false,
    pen pena, pair a,
    pen penb, pair b);

```

从以  $a$  圆心  $ra$  为半径的圆上的画笔  $pen_a$ ，到以  $b$  为圆心  $rb$  为半径的圆上的画笔  $pen_b$  的放射梯度光滑渐变，也是类似的：

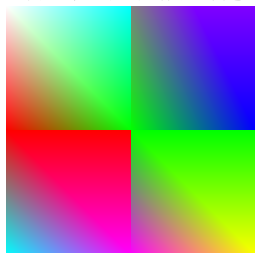
```

void radialshade(picture pic=currentpicture, path g, bool stroke=false,
    pen pena, pair a, real ra,
    pen penb, pair b, real rb);

```

<sup>6</sup>即这里填充的是具有宽度的路径本身。——译者注

<sup>7</sup>文档此处和实际行为不符，从早期版本开始，`latticeshade` 就一直会自动进行颜色空间的提升，从灰阶到 RGB 到 CMYK，因而  $p$  参数不需要手工提升到同一颜色空间。参看示例文件 `latticeshading.asy`：



```

size(200);
pen[] p={{white,greyscale,black},
    {red,green,blue},
    {cyan,magenta,yellow}};
latticeshade(unitsquare,p);

```

——译者注

<sup>8</sup>这里给一个例子：



```

unitsize(1cm);
path g = ellipse((0,0), 4, 1);
axialshade(g, green, (-4,0), yellow, (4,0));
axialshade(shift(0,-2.5)*g, stroke=true,
    green+linewidth(2mm), (-4,0), yellow, (4,0));

```

——译者注

放射渐变的例子在示例文件 `shade.asy`、`ring.asy` 和 `shadestroke.asy` 中<sup>9</sup>。

在由顶点集 `z` 和边标志 `edges` 定义的三角格点上, 使用填充规则 `fillrule` 及画笔数组 `p` 中的顶点颜色的 Gouraud 渐变, 由下面函数实现:

```
void gouraudshade(picture pic=currentpicture, path g, bool stroke=false,
    pen fillrule=currentpen, pen[] p, pair[] z,
    int[] edges);
void gouraudshade(picture pic=currentpicture, path g, bool stroke=false,
    pen fillrule=currentpen, pen[] p, int[] edges);
```

在第二种形式中, `z` 的元素取为依次路径 `g` 的各结点。 `p` 中的画笔必须属于同一颜色空间。 Gouraud 渐变的例子在示例文件 `Gouraud.asy` 和实体几何模块 `solids.asy` 中提供<sup>10</sup>。用于 Gouraud 渐变的边标志的文档见 <http://partners.adobe.com/public/developer/en/ps/sdk/TN5600.SmoothShading.pdf>。

在由路径数组 `b` 中的长度为 4 的  $n$  环形路径界定的补片上, 使用由  $n \times 4$  的画笔数组 `p` 和  $n \times 4$  数组内的中间控制点确定的顶点颜色, 使用填充规则 `fillrule` 的张量积渐变, 由下面函数实现:

```
void tensorshade(picture pic=currentpicture, path g, bool stroke=false,
    pen fillrule=currentpen, pen[][] p, path[] b=g,
    pair[][] z=new pair[][]);
```

如果数组 `z` 为空, 将使用 Coons 渐变, 其中颜色控制点会自动计算。 `p` 中画笔必须属于同一颜色空间。对只有单个补片 ( $n = 1$ ) 的情形的更简单的界面也是可用的:

```
void tensorshade(picture pic=currentpicture, path g, bool stroke=false,
    pen fillrule=currentpen, pen[] p, path b=g,
    pair[] z=new pair[]);
```

还可以使用给定的画笔数组, 光滑渐变填充连续一系列路径之间的区域:

```
void draw(picture pic=currentpicture, pen fillrule=currentpen, path[] g,
    pen[] p);
```

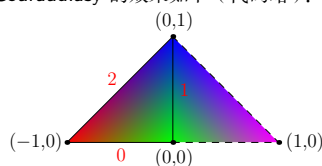
<sup>9</sup> 仅举 `shadestroke.asy` 一例:



```
size(100);
radialshade(W..N..E--(0,0),stroke=true,
    red+linewidth(30),(0,0),0.25,yellow,(0,0),1);
```

——译者注

<sup>10</sup> `p` 中画笔事实上会被自动提升到同一颜色空间。 `Gouraud.asy` 的效果如下 (代码略):



——译者注

张量积和 Coons 渐变的例子在示例文件 `tensor.asy`、`Coons.asy`、`BezierSurface.asy` 和 `rainbow.asy` 中<sup>11</sup>。

用 `pdfLATEX`、`ConTEXt` 和 `pdfTEX` 的 `TEX` 引擎可得到更一般的渐变的可能：过程

```
void functionshade(picture pic=currentpicture, path[] g, bool stroke=false,
                  pen fillrule=currentpen, string shader);
```

使用由字符串 `shader` 设定的 PostScript 计算过程，按填充规则 `fillrule`，在图 `pic` 上路径 `g` 内部的部分做渐变填充；该（PostScript）过程取 2 个参数，各自在  $[0, 1]$  取值，并返回 `colors(fillrule).length` 个颜色成分。函数式渐变在例子 `functionshading.asy` 中示例。

下面的过程使用 `evenodd` 剪裁连同 `^^` 运算符来反填充一个区域：

```
void unfill(picture pic=currentpicture, path g);
```

### 4.3 剪裁（clip）

```
void clip(picture pic=currentpicture, path g, stroke=false,
         pen fillrule=currentpen);
```

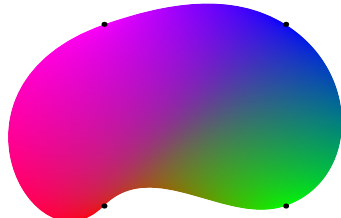
使用填充规则 `fillrule`（见 36 页 6.3 节），把图 `pic` 当前的内容按路径 `g` 界定的区域剪裁。如果 `stroke=true`，剪裁的部分与 `draw(pic,g,fillrule+zerowinding)` 绘制的区域相同；在这种情况下路径 `g` 不必是环形的。图剪裁的例子，见原手册第 6 章 `LATEX usage` 的第一个例子<sup>12</sup>。

### 4.4 标注（label）

```
void label(picture pic=currentpicture, Label L, pair position,
          align align=NoAlign, pen p=nullpen, filltype filltype=NoFill)
```

使用画笔 `p` 在图 `pic` 上绘制标签 `L`。如果 `align` 为 `NoAlign`，标签将在用户坐标 `position` 居中；否则将以 `align` 方向从 `position` 以 PostScript 坐标偏移 `align*labelmargin(p)` 的距离对齐。常数

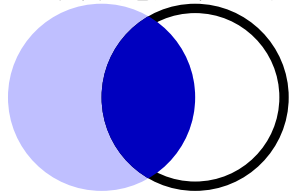
<sup>11</sup> 这里仅举出最简单的 Coons 渐变的例子：



```
size(200);
pen[] p={red,green,blue,magenta};
path g=(0,0){dir(45)}..(1,0)..(1,1)..(0,1)..cycle;
tensorshade(g,p);
dot(g);
```

——译者注

<sup>12</sup> 本译本未包含此章。这里另给一个简单的例子：



```
picture pic;
path C1 = circle((0,0), 1cm), C2 = circle((1cm,0), 1cm);
fill(C1, paleblue);          draw(pic, C2, linewidth(2mm));
fill(pic, C1, heavyblue);    clip(pic, C2);
add(pic);
```

——译者注



`Align` 可以用来按标签的左下角在 `position` 位置对齐。如果 `p` 为 `nullpen`，在标签中设定的画笔（默认为 `currentpen`）将被使用。标签 `L` 既可以是一个字符串也可以是通过调用如下函数之一得到的结构体：

```
Label Label(string s="", pair position, align align=NoAlign,
            pen p=nullpen, embed embed=Rotate, filltype filltype=NoFill);
Label Label(string s="", align align=NoAlign,
            pen p=nullpen, embed embed=Rotate, filltype filltype=NoFill);
Label Label(Label L, pair position, align align=NoAlign,
            pen p=nullpen, embed embed=L.embed, filltype filltype=NoFill);
Label Label(Label L, align align=NoAlign,
            pen p=nullpen, embed embed=L.embed, filltype filltype=NoFill);
```

标签的文字可以通过左乘一个仿射变换（见 46 页 6.4 节）而被放缩、倾斜、旋转或是平移。例如，`rotate(45)*xscale(2)*L` 首先在  $x$  方向放缩 `L` 而后将其逆时针方向旋转  $45^\circ$ 。标签的最终位置还可以通过 PostScript 坐标转换平移：`shift(10,0)*L`。`embed` 参数决定标签如何随着嵌入（此标签的）图变换：

#### Shift

仅随嵌入它的图平移；

#### Rotate

仅随嵌入它的图平移和旋转（默认值）；

#### Rotate(pair z)

按（随图变换过的）向量 `z` 旋转；

#### Slant

仅随嵌入它的图平移、旋转、倾斜、反射；

#### Scale

随嵌入它的图的平移、旋转、倾斜、反射和放缩；

要给一条路径加标签，用

```
void label(picture pic=currentpicture, Label L, path g, align align=NoAlign,
          pen p=nullpen, filltype filltype=NoFill);
```

标签默认将位于路径的中点。另一种标签位置（在 `point(path p, real t)` 意义下）可以在构造标签时用一个实数值 `position` 设定。位置 `Relative(real)` 确定一个相对于路径总弧长的位置<sup>13</sup>。如下的方便缩写形式已经预定义好了：

<sup>13</sup> 因而除了上面的提及的四种 `Label` 构造函数外，还有两种接受 `position` 类型参数的构造函数：

```
Label Label(string s, string size="", explicit position position,
            align align=NoAlign, pen p=nullpen, embed embed=Rotate,
            filltype filltype=NoFill);
Label Label(Label L, explicit position position, align align=NoAlign,
```

```
position BeginPoint=Relative(0);
position MidPoint=Relative(0.5);
position EndPoint=Relative(1);
```

路径标签以方向 `align` 对齐，它可由一个绝对罗盘方向（复数 `pair`）或相对于路径本地方向北极轴测量的方向 `Relative(pair)` 设定<sup>14</sup>。为方便计，`LeftSide`、`Center` 和 `RightSide` 分别定义为 `Relative(W)`、`Relative((0,0))` 以及 `Relative(E)`。给 `LeftSide`、`Center` 和 `RightSide` 左乘一个实数放缩因子会把标签移得更靠近或更远离路径<sup>15</sup>。

带有从方向 `dir` 来指向 `b` 的长为 `arrowlength` 的固定尺寸箭头的标签，可以由如下过程产生：

```
void arrow(picture pic=currentpicture, Label L="", pair b, pair dir,
    real length=arrowlength, align align=NoAlign,
    pen p=currentpen, arrowbar arrow=Arrow, margin margin=EndMargin);
```

如果没有设定对齐方向（不论是在标签中还是明确以参数给出），可选的标签将使用边距 `align`，以 `dir` 的方向对齐。

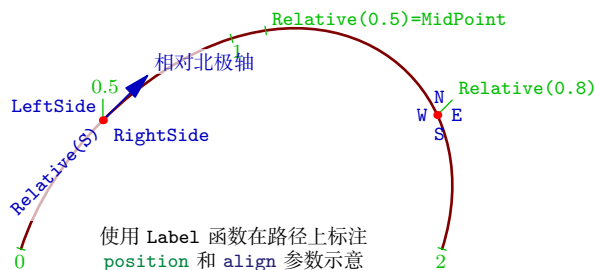
函数 `string graphic(string name, string options="")` 返回一个可以用于插入嵌入式 PostScript（Encapsulated PostScript, EPS）文件的字符串。这里，`name` 是要插入的文件名而 `options` 是一个逗号分隔的参数列表字符串：包括可选的边框（`bb=llx lly urx ury`）、宽度（`width=value`）、高度（`height=value`）、旋转（`angle=value`）、放缩（`scale=factor`）、剪裁（`clip=bool`）以及草稿模式（`draft=bool`）。`layer()` 函数可用来强迫将来的对象在插入的图像上方绘制：

```
label(graphic("file.eps","width=1cm"),(0,0),NE);
layer();
```

```
pen p=nullpen, embed embed=L.embed, filltype filltype=NoFill);
```

其 `position` 参数即可以是一个实数，也可以是一个 `Relative(value)` 值。——译者注

<sup>14</sup>`Relative(N)` 就是路径在此位置的切向。有关这种 `Label` 的 `position` 和 `align` 参数的意义，如图所示：



——译者注

<sup>15</sup>除了上文和脚注 13 中的六种 `Label` 构造函数，在 `plain` 模块中还有一种构造函数，它将标签位置（复数 `pair` 类型）自动转换为标签的内容：

```
Label Label(pair position, align align=NoAlign, pen p=nullpen,
    embed embed=Rotate, filltype filltype=NoFill);
```

——译者注

`string baseline(string s, string template="\strut")` 函数可用于扩大标签的边框以匹配给定的模板，从而这些标签的基线将排在同一水平线上。参见 `Pythagoras.asy` 的例子<sup>16</sup>。

用 `overwrite` 画笔属性（参看 36 页 6.3 节），可以避免标签互相覆写。

在 `plain_Label.asy` 中定义的结构体 `object`（对象）允许以一致的行为方式看待标签和帧。用如下过程可以将一组对象打包进一帧内：

```
frame pack(pair align=2S ... object inset[]);
```

要绘制或填充围绕一个标签的方盒（或是椭圆或其他路径）并返回界定的对象，使用如下过程之一：

```
object draw(picture pic=currentpicture, Label L, envelope e,
    real xmargin=0, real ymargin=xmargin, pen p=currentpen,
    filltype filltype=NoFill, bool above=true);
object draw(picture pic=currentpicture, Label L, envelope e, pair position,
    real xmargin=0, real ymargin=xmargin, pen p=currentpen,
    filltype filltype=NoFill, bool above=true);
```

这里 `envelope` 是在 `plain_boxes.asy` 中定义的边界绘制过程（参看 27 页 6.2 节），如 `box`、`roundbox` 或 `ellipse`。

函数 `path[] texpath(Label L)` 返回  $\text{T}_{\text{E}}\text{X}$  将要为绘制标签 `L` 而填充的路径。

函数 `string minipage(string s, width=100pt)` 可用于将字符串 `s` 格式化为一个宽度为 `width` 的段落。这个例子利用 `minipage`、`clip` 和 `graphic` 来生成一个 CD 标签：

---

<sup>16</sup>`Pythagoras.asy` 的例子见脚注 4。——译者注



```
size(11.7cm,11.7cm);
asy(nativeformat(),"logo");
fill(unitcircle^^(scale(2/11.7)*unitcircle),
      evenodd+rgb(124/255,205/255,124/255));
label(scale(1.1)*minipage(
  "\centering\scriptsize \textbf{\LARGE {\tt Asymptote}}\\
  \smallskip
  \small The Vector Graphics Language}\\
  \smallskip
  \textsc{Andy Hammerlindl, John Bowman, and Tom Prince}
  http://asymptote.sourceforge.net\\
  ",8cm),(0,0.6));
label(graphic("logo."+nativeformat(),"height=7cm"),(0,-0.22));
clip(unitcircle^^(scale(2/11.7)*unitcircle),evenodd);
```

## 第五章 Bézier 曲线

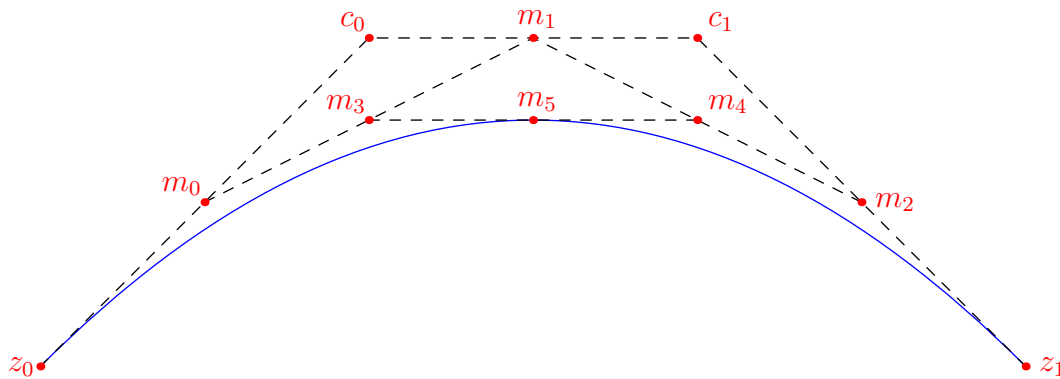
三次样条曲线的每个内结点可以指定一个前缀或后缀的方向  $\{\text{dir}\}$ : 复数  $\text{dir}$  的方向分别设定该曲线在此结点处, 入向或出向切线的方向。外结点仅能在向内一侧指定限定方向。

在带有后控制点  $c_0$  的结点  $z_0$ , 与带有前控制点  $c_1$  的结点  $z_1$  之间的一段三次样条曲线, 按 Bézier 曲线

$$(1-t)^3 z_0 + 3t(1-t)^2 c_0 + 3t^2(1-t) c_1 + t^3 z_1, \quad 0 \leq t \leq 1$$

计算<sup>1</sup>。

如下图所示, 由两个终点  $z_0$  和  $z_1$  以及两个控制点  $c_0$  与  $c_1$  构造的三阶中点 ( $m_5$ ), 正是在由四元组  $(z_0, c_0, c_1, z_1)$  建立的 Bézier 曲线上对应于  $t = 1/2$  的点。这就允许人们通过将重新提取的三阶中点用作终点, 并将二阶与一阶中点用作控制点, 递归地构造想要的曲线:



这里  $m_0, m_1$  和  $m_2$  是一阶中点,  $m_3$  与  $m_4$  是二阶中点, 而  $m_5$  是三阶中点。于是曲线就通过对  $(z_0, m_0, m_3, m_5)$  和  $(m_5, m_4, m_2, z_1)$  递归应用此算法构造出来。

事实上, 类似的性质对于每小段中在  $[0, 1]$  上的任意分数  $t$  位置的点都成立, 而不仅仅是中点 ( $t = 1/2$ )。

用这种方式构造的 Bézier 曲线具有如下性质:

它完全包含在给定四个点的凸包内。

其起点朝向从第一个端到第一个控制点而其终点朝向从第二个控制点到第二个端点。

<sup>1</sup>本节关于三次 Bézier 样条曲线的讲解是比较难懂的, 并且下图中关于曲线离散生成的知识事实上与绘图无关。关于曲线控制的更清晰详细的介绍可以参考 METAPOST 的手册 *METAPOST: A User's Manual* 第 4 章 Curves, 以及关于 METAPOST 与 ConTeXt 的文档 *METAPOST* 的第 1 章。——译者注

用户可以像这样设定两个结点间的显式控制点：

```
draw((0,0)..controls (0,100) and (100,100)..(100,0));
```

然而，通常更为方便的是仅使用 .. 运算符，它告诉 Asymptote 使用在 Donald Knuth 的专著 *The METAFONTbook* 的第 14 章中描述的算法来选择自己的控制点。用户还可以通过设定方向、张力和卷曲值来定制路向（或路径）。

张力越强，曲线越直，它也就越接近直线。可以将样条曲线的张力从其默认值 1 变为任意大于等于 0.75 的实数值（参考 John D. Hobby, *Discrete and Computational Geometry 1*, 1986）：

```
draw((100,0)..tension 2 ..(100,100)..(0,100));
draw((100,0)..tension 2 and 1 ..(100,100)..(0,100));
draw((100,0)..tension atleast 1 ..(100,100)..(0,100));
```

卷曲参数设定在路径端点处的弯曲程度（0 表示直的；默认值 1 表示近似为圆）：

```
draw((100,0){curl 0}..(100,100)..{curl 0}(0,100));
```

METAPOST 的 ... 路径连接符，在可能时它要求一个限定在由端点和方向定义的三角形内的无拐点的曲线，它在 Asymptote 中实现为 .. tension atleast 1 .. 的方便缩写 ::（省略号 ... 在 Asymptote 中用于标示可变长参数；见 69 页 6.11.3 节剩余参数）。例如，比较

```
draw((0,0){up}..(100,25){right}..(200,0){down});
```



与

```
draw((0,0){up}::(100,25){right}::(200,0){down});
```



--- 连接符是 ..tension atleast infinity.. 的缩写<sup>2</sup>，而 & 连接符连结两条路径，并去除第一条路径的第最后一个结点（它一般应该与第二条路径的第一个结点重合）。

<sup>2</sup>它的效果和通常的折线连接基本相同，只是在弯折处比完全的折线更为圆滑。——译者注

## 第六章 编程

这里是对程序设计语言 Asymptote 的一个介绍性的例子，强调其中与 C, C++ 和 Java 相似的控制结构<sup>1</sup>。

```
// 这是一行注释。

// 声明：声明 x 是一个 real 变量；
real x;

// 赋值：给 real 变量 x 赋值 1 。
x=1.0;

// 条件：测试 x 是否等于 1 。
if(x == 1.0) {
    write("x 等于 1.0");
} else {
    write("x 不等于 1.0");
}

// 循环：迭代 10 次
for(int i=0; i < 10; ++i) {
    write(i);
}
```

正如在 C/C++ 中一样，Asymptote 支持 `while`, `do`, `break` 和 `continue` 语句。Java 风格的迭代遍历数组元素的简便形式也是支持的：

```
// 迭代遍历一个数组
int[] array={1,1,2,3,5};
for(int k : array) {
    write(k);
}
```

---

<sup>1</sup>这个文档无疑是面向具有 C, C++ 或 Java 等语言编程经验的人的，因而这里关于控制流的说明简略得甚至有些过分，比如没有给出 `do` 结构的例子。有需要的读者请参考相关语言和程序设计的其他书籍。——译者注

```
}
```

此外，它也支持许多上述语言之外的特征。

## 6.1 数据类型

Asymptote 支持下列数据类型（除用户自定义类型外）：

**void** 空（**void**）类型仅用于无参数或无返回值的函数。

**bool** 布尔类型只能取值 **true** 或 **false**。例如：

```
bool b = true;
```

定义一个布尔变量 **b** 并初始化为值 **true**。如果没有给定初始式：

```
bool b;
```

其值假定为 **false**。

**bool3** 一种扩展的布尔类型，可以取值 **true**, **default** 或 **false**。**bool3** 类型可以与 **bool** 类型相互转换。**bool3** 的默认初始式为 **default**。

**int** 整数类型；如果没有给定初始式，隐式的值假定为 0。整数允许的最小值为 **intMin**，最大值为 **intMax**。

**real** 实数；它将设为计算机结构的本地浮点类型的最大精度。实数的隐式初始式为 0.0。实数具有精度 **realEpsilon**，有效数字为 **realDigits**。最小的正实数为 **realMin**，而最大的正实数为 **realMax**。

**pair** 复数，即，实数构成的有序对 (**x**, **y**)。复数 **z** 的实部和虚部可读为 **z.x** 和 **z.y**。我们称 **x** 和 **y** 为复数数据元素的虚拟成员；然而，它们不能直接修改。复数的隐式初始式为 (0.0,0.0)。

有许多方法取复数的复共轭：

```
pair z=(3,4);
z=(z.x,-z.y);
z=z.x-I*z.y;
z=conj(z);
```

这里 **I** 是复数 (0,1)。复数类型定义了许多内部函数：

```
pair conf(pair z)
```

返回 **z** 的复共轭。

```
real length(pair z)
```

返回参数 **z** 的复数模长  $|z|$ 。例如，



```
pair z=(3,4);  
length(z);
```

返回结果 5。`abs(pair)` 与 `length(pair)` 同义;

```
real angle(pair z, bool warn=true)
```

返回  $z$  的幅角, 单位为弧度, 在区间  $[-\pi, \pi]$  内, 而当 `warn` 为 `false` 且  $z = (0,0)$  时返回 0 (而不是产生错误);

```
real degrees(pair z, bool warn=true)
```

返回  $z$  的幅角, 单位为度, 在区间  $[0, 360)$  内, 而当 `warn` 为 `false` 且  $z = (0,0)$  时返回 0 (而不是产生错误);

```
pair unit(pair z)
```

返回以复数  $z$  同方向的单位向量;

```
pair expi(real angle)
```

返回以弧度角 `angle` 为方向的单位向量;

```
pair dir(real degrees)
```

返回以角度 `degrees` 为方向的单位向量;

```
real xpart(pair z)
```

返回  $z.x$ ;

```
real ypart(pair z)
```

返回  $z.y$ ;

```
pair realmult(pair z, pair w)
```

返回逐元素乘积  $(z.x*w.x, z.y*w.y)$ ;

```
real dot(pair z, pair w)
```

返回点积  $z.x*w.x + z.y*w.y$ ;

```
pair minbound(pair z, pair w)
```

返回  $(\min(z.x, w.x), \min(z.y, w.y))$ ;

```
pair maxbound(pair z, pair w)
```

返回  $(\max(z.x, w.x), \max(z.y, w.y))$ 。

`triple` 有序实数三元组（向量） $(x, y, z)$  用于三维作图。三元组  $v$  的对应组成部分分别读为  $v.x$ ,  $v.y$  和  $v.z$ 。隐式初始式为  $(0.0, 0.0, 0.0)$ 。

这里是三元组的内部函数：

```
real length(triple v)
```

返回向量  $v$  的长度  $|v|$ 。`abs(triple)` 与 `length(triple)` 同义；

```
real polar(triple v, bool warn=true)
```

返回  $v$  的余纬角，从  $z$  轴计，单位为弧度，而当 `warn` 为 `false` 且  $v = 0$  时<sup>2</sup>返回 0（而不是产生错误）；

```
real azimuth(triple v, bool warn=true)
```

返回  $v$  的经度角，从  $x$  轴计，单位为弧度，而当 `warn` 为 `false` 且  $v.x = v.y = 0$  时返回 0（而不是产生错误）；

```
real colatitude(triple v, bool warn=true)
```

返回  $v$  的余纬角，从  $z$  轴计，单位为度，而当 `warn` 为 `false` 且  $v = 0$  时返回 0（而不是产生错误）；

```
real latitude(triple v, bool warn=true)
```

返回  $v$  的纬度角，从  $xy$  平面计，单位为度，而当 `warn` 为 `false` 且  $v = 0$  时返回 0（而不是产生错误）；

```
real longitude(triple v, bool warn=true)
```

返回  $v$  的经度角，从  $x$  轴计，单位为度，而当 `warn` 为 `false` 且  $v.x = v.y = 0$  则返回 0（而非产生错误）；

```
triple unit(triple v)
```

返回与三元组（向量） $v$  同方向的单位向量；

```
triple expi(real polar, real azimuth)
```

返回方向为  $(polar, azimuth)$ （单位为弧度）的单位向量；

```
triple dir(real colatitude, real longitude)
```

返回方向为  $(colatitude, longitude)$ （单位为度）的单位向量；

```
real xpart(triple v)
```

返回  $v.x$ ；

---

<sup>2</sup>即  $v$  的所有坐标均为 0。0 不是内部的常量，而是在 `three` 模块中定义的变量  $(0, 0, 0)$ 。后同。——译者注

```

real ypart(triple v)
    返回 v.y;

real zpart(triple v)
    返回 v.z;

triple realmult(triple u, triple v)
    返回逐元素乘积3 (u.x*v.x, u.y*v.y, u.z*v.z);

real dot(triple u, triple v)
    返回点积 u.x*v.x+u.y*v.y+u.z*v.z;

triple cross(triple u, triple v)
    返回叉积 (u.y*v.z-u.z*v.y,u.z*v.x-u.x*v.z,u.x*v.y-v.x*u.y);

triple minbound(triple u, triple v)
    返回 (min(u.x,v.x),min(u.y,v.y),min(u.z,v.z));

triple maxbound(triple u, triple v)
    返回 (max(u.x,v.x),max(u.y,v.y),max(u.z,v.z))。

```

`string` 字符串，它使用 STL 的 `string` 类实现。

用双引号 (") 界定的字符串受限于下列映射，以允许在  $\text{T}_{\text{E}}\text{X}$  中双引号的使用（如为了使用 `babel` 包，见文档 7.21 节 `babel`<sup>4</sup>）：

- \" 映射到 "
- \\ 映射到 \

用单引号 (') 界定的字符串具有与 ANSI C 相同的字符映射：

- \' 映射到 '
- \" 映射到 "
- \? 映射到 ?
- \\ 映射到反斜线
- \a 映射到响铃
- \b 映射到退格
- \f 映射到换页符

<sup>3</sup>这个函数在原手册中缺失，这里根据源代码加上。——译者注

<sup>4</sup>本译本不包括此部分。——译者注

- `\n` 映射到换行符
- `\r` 映射到回车符
- `\t` 映射到制表符
- `\v` 映射到竖直制表符
- `\0-\377` 映射到对应的八进制字节
- `\x0-\xFF` 映射到对应的十六进制字节<sup>5</sup>

字符串的隐式初始式为空串 ""。字符串可以用运算符 + 拼接。在下列字符串函数中，位置 0 表示字符串的开始：

`int length(string s)`

返回字符串 `s` 的长度；

`int find(string s, string t, int pos=0)`

返回字符串 `t` 在字符串 `s` 在位置 `pos` 处或之后，第一次出现的位置，或是在 `t` 不是 `s` 的子串时返回 -1；

`int rfind(string s, string t, int pos=-1)`

返回字符串 `t` 在字符串 `s` 在位置 `pos` 处或之前（若 `pos = -1`，则在字符串 `s` 末尾），最后一次出现的位置，或是在 `t` 不是 `s` 的子串时返回 -1；

`string insert(string s, int pos, string t)`

返回将字符串 `t` 插入 `s` 位置 `pos` 处构成的字符串；

`string erase(string s, int pos, int n)`

返回将字符串 `s` 在位置 `pos` 处删去长度为 `n` 的串（如果 `n = -1`，删至字符串 `s` 末尾）构成的字符串；

`string substr(string s, int pos, int n=-1)`

返回从位置 `pos` 开始长度为 `n`（如果 `n = -1`，直至字符串 `s` 的末尾）的子串；

`string reverse(string s)`

返回由字符串 `s` 翻转构成的字符串；

`string replace(string s, string before, string after)`

返回将字符串 `s` 中所有串 `before` 的出现改为串 `after` 得到的字符串；

---

<sup>5</sup>在这里十六进制的字母须大写，而 C 语言的转义符可用小写。另外，`Asymptote` 并没有像 C 中那样八进制和十六进制整数类型的字面量（如 `0177`、`0x2F`）。——译者注

```
string replace(string s, string[] [] table)
```

返回将字符串 `s` 中将数组 `table` 中字符串对 `{before,after}` 中的串 `before` 的所有出现转换为对应的串 `after` 构成的字符串；

```
string[] split(string s, string delimiter)
```

返回一个字符串数组，包含将 `s` 用 `delimiter` 定界分割而成的子串；

```
string format(string s, int n)
```

返回使用当前区域设置 (`locale`)，将 `n` 用 C 风格格式化串格式化得到的字符串<sup>6</sup>；

```
string format(string s=defaultformat, real x, string locale="")
```

返回使用区域设置 `locale` (或当前区域设置，如果设定为空串)，按照 C 函数 `fprintf` 的行为方式用 C 风格格式化串格式化 `x` 得到的字符串，除了只允许一个数据域时，末尾的零默认会被删去 (除非设定了 `#`)，并且 (若格式串设定了数学模式) `TeX` 将用于排版科学计数法<sup>7</sup>；

```
int hex(string s)
```

将十六进制字符串 `s` 转换为一个整数。

```
string string(real x, int digits=realDigits)
```

使用精度 `digits` 和 C 区域设置 (`locale`) 将 `x` 转换为字符串；

```
string locale(string s="")
```

设定区域设置为给定的字符串 (如果此字符串非空)，并返回当前的区域设置；

```
string time(string format="%a %b %d %T %Z %Y")
```

返回用 ANSI C 过程 `strftime`，依字符串 `format` 和当前区域设置格式化的当前时间<sup>8</sup>。因而

```
time();
time("%a %b %d %H:%M:%S %Z %Y");
```

<sup>6</sup>例如，`format("%04d", 3)` 得到字符串 "0003"。由于 `Asymptote` 中只有一种整数类型，所以表示整数长度的修饰符 `h`、`l`、`ll` 是无用的。可用的格式符主要为 `%d` (或 `%i`，十进制整数)、`%o` (八进制)、`%x` (十六进制小写)、`%X` (十六进制大写)、`%u` (无符号十进制整数)，字母前可用最小宽度、前导零、`#` 和正负号修饰。这里文档未提及的 `#` 修饰符表示输出八进制和十六进制数时同时输出 `0` 或 `0x` (`0X`) 前缀，正号前缀表示对正数也总输出正号 (负数不变)，负号前缀表示左对齐 (在限定最小输出宽度时有用)。当然，这里不支持 `%*d` 这种具有不同函数原型的复杂语法 (其功能可以用 `format("%"+string(4)+"d", 3)` 这种用法代替)。——译者注

<sup>7</sup>`Asymptote` 中数字没有千分位，区域设置通常主要会影响小数点 (有些区域设置使用逗号)，不过译者在 Windows 版本下没能成功改变这个设置。表示双精度的 `L` 是无用的。可用的格式符主要为 `%f` (纯小数形式)、`%e` (科学计数法)、`%g` (自动选取纯小数和科学计数法中相对较短的形式)，字符前可用最小宽度、精度、前导零、`#` 和正负号 (作用与整数 `format(string, int)` 的参数相同) 来修饰。注意从 1.77 版开始引入的对数学模式的判断并不准确，函数只是简单地搜索格式串中是否有字符 `$`。使用格式串修饰符的例子如：`format("%010.5f", pi)` 得到字符串 "0003.14159"，`format("%+.3g", 0.00001234)` 得到字符串 "+1.23e-05"，而 `format("$%+.3g$", 0.00001234)` 则得到 "\$+1.23\!\times\!10^{-5}\$"。——译者注

<sup>8</sup>如下日期格式串的格式化字符串表格参考 K&R 和 C99 标准中的描述，并据实际测试和后面文档的说明略有改动：

是返回以 Unix date 命令默认格式化的当前时间的等价的方法；

```
int seconds(string t="", string format="")
```

返回新纪元（UTC 1970 年 1 月 1 日星期四，00:00:00）之后到某一时间的秒数。此时间由 ANSI C 过程 `strptime` 使用当前区域设置按字符串 `format` 确定，或在 `t` 为空串时取当前时间。注意 POSIX `strptime` 说明的 "%Z" 扩展在目前的 GNU C 库中被忽略。如果产生错误，返回值 -1。这里是一些例子：

```
seconds("Mar 02 11:12:36 AM PST 2007", "%b %d %r PST %Y");
seconds(time("%b %d %r %Z %Y"), "%b %d %r %Z %Y");
seconds(time("%b %d %r %Z %Y"), "%b %d %r "+time("%Z")+ " %Y");
1+(seconds()-seconds("Jan 1", "%b %d"))/(24*60*60);
```

最后一个例子返回从元旦开始，今天的日期序数<sup>9</sup>。

```
string time(int seconds, string format="%a %b %d %T %Z %Y")
```

返回对应于新纪元（UTC 1970 年 1 月 1 日星期四，00:00:00）之后 `seconds` 秒的时间，由 ANSI C 过程 `strftime` 使用当前区域设置按字符串 `format` 格式化。例如，要返回对应于 24 小时之前的日期：

```
time(seconds()-24*60*60);
```

```
void abort(string s)
```

中止执行（在批处理模式时返回一个非零值）；如果字符串 `s` 非空，则会打印由源文件名、行号和 `s` 构成的诊断信息。

%a	缩写的星期名。	%A	全写的星期名。
%b	缩写的月份名。	%B	全写的月份名。
%c	本地（当前区域设置）的日期与时间表示。	%C	年份的世纪部分（年份除以 100 并取整的结果）。
%d	一月内的日期（01-31）。	%D	等价于 %m/%d/%y。
%e	一月内的日期（1-31）；一位数前面用空格补齐。	%F	等价于 %Y-%m-%d（此为 ISO 8601 日期格式）。
%g	ISO 8601 基于星期的年份的后两位（00-99）。	%G	基于星期的年份。
%h	等价于 %b。	%H	小时（24 小时制）（00-23）。
%I	小时（12 小时制）（01-12）。	%j	一年内的天数（001-366）。
%m	月份（01-12）。	%M	分钟（00-59）。
%n	换行符。	%p	AM 或 PM 的本地等价物。
%r	本地的 12 小时制时间。	%R	等价于 %H:%M。
%S	秒（00-60）。	%t	水平制表符。
%T	等价于 %H:%M:%S（此为 ISO 8601 时间格式）。	%u	星期（1-7），其中周一是 1。
%U	一年内的星期数（周日是每星期的第一天）（00-53）。	%w	星期（0-6，周日是 0）。
%W	一年内的星期数（周一是每星期的第一天）（00-53）。	%V	ISO 8601 星期数（01-53）。
%x	本地的日期表示。	%X	本地的时间表示。
%y	不含世纪的年（00-99）。	%Y	含世纪的年。
%z	当前时间与 UTC（协调世界时）的差距，用 ISO 8601 格式表示。例如 "+0800" 表示比格林威治西部的 UTC 时间大（早）8 小时 0 分。若时区不确定则没有字符。	%%	符号 % 本身。

以上只是简略解释，此外还有一些复合形式。更多详情请参看相关手册。——译者注

<sup>9</sup>相当于 `(int)time("%j")`。——译者注

```
void exit()
```

在批处理模式以错误返回值零退出<sup>10</sup>。

```
void sleep(int seconds)
```

暂停给定的秒数。

```
void usleep(int microseconds)
```

暂停给定的毫秒数。

```
void beep()
```

在控制台产生一声蜂鸣 (哔声)。

像在 C/C++ 中一样, 复合类型可以用 `typedef` 缩写<sup>11</sup> (见 65 页 6.11 节的例子)。

## 6.2 路径 (path) 与路向 (guide)

`path` 解析为确定路径的三次样条。路径的隐式的初始式为 `nullpath`。

例如, 过程 `circle(pair c, real r)`, 返回一条近似为原点在 `c` 半径为 `r` 的圆的 Bézier 曲线, 它基于 `unitcircle` (见 1 页第三章<sup>12</sup>):

```
path circle(pair c, real r)
{
    return shift(c)*scale(r)*unitcircle;
}
```

如果需要高精度, 可以用模块 `graph.asy` 中定义的 `Circle` 过程来产生真圆<sup>13</sup>:

```
import graph;
path Circle(pair c, real r, int n=nCircle);
```

与 `circle` 一致, 圆心为 `c` 半径为 `r`, 从 `angle1` 到 `angle2` 度, 在  $\text{angle2} \geq \text{angle1}$  时逆时针绘制的圆弧, 可以这样构造:

```
path arc(pair c, real r, real angle1, real angle2);
```

<sup>10</sup>程序退出时返回值为零即为无错误。——译者注

<sup>11</sup>但 `typedef` 定义的类型缩写对函数类型有一定限制: 它不能用于 `new` 之后构成匿名函数, 其中也不能有函数的默认参数 (注意默认参数是在函数定义时求值的)。——译者注

<sup>12</sup>`unitcircle` 的定义为

```
path unitcircle=E..N..W..S..cycle;
```

即使用 4 个节点的闭三次样条曲线近似表示的单位圆路径, 起点在 (1,0) 点, 方向为逆时针方向。——译者注

<sup>13</sup>即返回结点数为 `nCircle` 的路径来逼近圆。在 `graph` 模块中, `nCircle` 的默认值是 400, 实用上这个数值太大了——原来四个结点的曲线仅和精确的圆相差不足 0.06%, 太多的结点在输出设备上体现不出来, 却要大大增加绘制、填充等路径操作的计算量。因此译者建议实用中先重定义 `nCircle` 为较小的数值。——译者注

也可以明确指定方向：

```
path arc(pair c, real r, real angle1, real angle2, bool direction);
```

这里方向 `direction` 可以指定为 `CCW` (counter-clockwise, 逆时针) 或 `CW` (clockwise, 顺时针)。为方便计, 从复数 (点) `z1` 到 `z2`, 圆心为 `c` 的圆弧 (假定<sup>14</sup>  $|z2 - c| = |z1 - c|$ ) 也可以这样构造:

```
path arc(pair c, explicit pair z1, explicit pair z2,
         bool direction=CCW)
```

如果需要高精度, 可以用模块 `graph.asy` 中定义的过程来产生 `n` 个结点的 Bézier 曲线真圆弧:

```
import graph;
path Arc(pair c, real r, real angle1, real angle2, bool direction,
        int n=nCircle);
path Arc(pair c, real r, real angle1, real angle2, int n=nCircle);
path Arc(pair c, explicit pair z1, explicit pair z2,
        bool direction=CCW, int n=nCircle);
```

椭圆可以用此过程绘制:

```
path ellipse(pair c, real a, real b)
{
    return shift(c)*scale(a,b)*unitcircle;
}
```

这个例子说明了在手册第三章教程讨论的全部五种路向连接:

```
size(300,0);
pair[] z=new pair[10];

z[0]=(0,100); z[1]=(50,0); z[2]=(180,0);

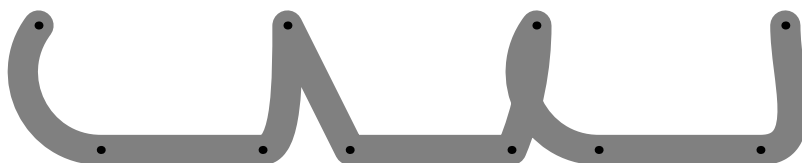
for(int n=3; n <= 9; ++n)
    z[n]=z[n-3]+(200,0);

path p=z[0]..z[1]---z[2>::{up}z[3]
&z[3]..z[4]--z[5>::{up}z[6]
&z[6>::z[7]---z[8]..{up}z[9];
```

<sup>14</sup>从 `plain` 模块源码来看, 这个函数实际得到的是圆心在 `c`, 起点在 `z1`, 圆心到终点的方向为 `z2 - c` 的圆弧。即半径仅由 `c` 和 `z1` 确定。——译者注



```
draw(p, grey+linewidth(4mm));
dot(z);
```



这里是一些对路径有用的函数：

```
int length(path p);
```

这是路径  $p$  的（直线或三次曲线）段数。如果  $p$  是环形，这就与  $p$  的结点数相同。

```
int size(path p);
```

这是路径  $p$  的结点数。如果  $p$  是环形，这就与 `length(p)` 相同。

```
bool cyclic(path p);
```

返回 `true` 当且仅当路径  $p$  为环形<sup>15</sup>。

```
bool straight(path p, int i);
```

返回 `true` 当且仅当路径  $p$  在结点  $i$  和结点  $i+1$  之间的线段是直的。

```
bool piecewisestraight(path p);
```

返回 `true` 当且仅当路径  $p$  是分段直的（即为折线）。

```
pair point(path p, int t);
```

如果  $p$  是环形，返回结点  $t \bmod \text{length}(p)$  的坐标。否则，返回结点  $t$  的坐标，除非  $t < 0$ （此时返回 `point(0)`）或  $t > \text{length}(p)$ （此时返回 `point(length(p))`）。

```
pair point(path p, real t);
```

这将返回结点 `floor(t)` 与 `floor(t)+1` 之间对应于三次样条参数  $t - \text{floor}(t)$  的点的坐标（参看手册第三章教程 Bézier 曲线部分）。如果  $t$  在  $[0, \text{length}(p)]$  的范围之外，则在  $p$  是环形的情况下首先模 `length(p)` 约化或在  $p$  非环形的情况下转化为  $p$  的对应终点。

```
pair dir(path p, int t, int sign=0, bool normalize=true);
```

<sup>15</sup>即封闭路径。——译者注

如果 `sign < 0`，则返回路径 `p` 在结点 `t` 处的入向切线方向（作为一个复数）；如果 `sign > 0`，则返回出向切线方向。如果 `sign = 0`，则返回两个方向的平均值。

```
pair dir(path p, real t, bool normalize=true);
```

返回路径 `p` 在结点 `floor(t)` 和 `floor(t)+1` 之间对应于三次样条参数 `t-floor(t)` 的点的切线方向（参看手册第三章教程，Bézier 曲线部分）。

```
pair accel(path p, int t, int sign=0);
```

如果 `sign < 0`，则返回路径 `p` 在结点 `t` 处的入向加速度（作为一个复数）；如果 `sign > 0`，则返回出向加速度。如果 `sign = 0`，则返回两个加速度的平均值。

```
pair accel(path p, real t);
```

返回路径 `p` 在点 `t` 处的加速度。

```
pair radius(path p, real t);
```

返回路径 `p` 在点 `t` 处的半径<sup>16</sup>。

```
pair precontrol(path p, int t);
```

返回 `p` 在结点 `t` 处的前控制点。

```
pair precontrol(path p, real t);
```

返回 `p` 在参数 `t` 处的有效前控制点。

```
pair postcontrol(path p, int t);
```

返回 `p` 在结点 `t` 处的后控制点。

```
pair postcontrol(path p, real t);
```

返回 `p` 在结点 `t` 处的有效后控制点。

```
real arclength(path p);
```

返回路径 `p` 表示的分段线性或三次曲线的长度（按用户坐标）。

```
real arctime(path p, real L);
```

返回路径的“时刻<sup>17</sup>”，一个在 0 和路径长度之间，在 `point(path p, real t)` 意义下的实数，在此时刻累积弧长（从路径开始处度量）等于 `L`。

```
real dirstime(path p, pair z);
```

返回第一个“时刻”，它是在 0 和路径长度之间，在 `point(path, real)` 意义下的实数，在此时刻路径的切线具有复数 `z` 的方向，否则若没有这样的时刻则返回 `-1`。

<sup>16</sup>指曲率半径，即密切圆的半径。——译者注

<sup>17</sup>指曲线的时间参数。——译者注

```
real reltime(path p, real l);
```

返回路径  $p$  在其弧长的相对分数  $l$  处的时刻。

```
pair relpoint(path p, real l);
```

返回路径  $p$  在其弧长的相对分数  $l$  处的点。

```
pair midpoint(path p);
```

返回路径  $p$  在其一半弧长处点。

```
path reverse(path p);
```

返回一条沿  $p$  反向行进的路径。

```
path subpath(path p, int a, int b);
```

返回  $p$  从结点  $a$  到结点  $b$  行进的子路径。如果  $a < b$ , 则子路径的方向会反向。

```
path subpath(path p, real a, real b);
```

返回  $p$  从路径时刻  $a$  到路径时刻  $b$  行进的子路径。如果  $a < b$ , 则子路径的方向会反向。

```
real[] intersect(path p, path q, real fuzz=-1);
```

如果  $p$  与  $q$  至少有一个交点, 则返回一个长度为 2 的实数数组, 它包含表示对于这样一个交点 (交点按 *The METAFONTbook* 137 页描述的算法选择<sup>18</sup>), 在 `point(path p, real t)` 意义下, 分别沿  $p$  和  $q$  的路径时刻。计算以 `fuzz` 确定的绝对误差执行, 或者如果 `fuzz < 0`, 则达到机器精度。如果路径不相交, 返回一个长度为 0 的实数数组。

```
real[] [] intersections(path p, path q, real fuzz=-1);
```

以排序的 (参看 72 页 6.12 节排序的介绍), 长度为 2 的实数数组的数组, 返回路径  $p$  和  $q$  的所有 (除非有无穷多个) 相交时刻。计算以 `fuzz` 确定的绝对误差进行, 或者如果 `fuzz < 0`, 则达到机器精度。

```
real[] intersections(path p, explicit pair a, explicit pair b, real fuzz=-1);
```

以已排序数组返回路径  $p$  与通过  $a$ 、 $b$  两点的 (无穷) 直线的所有 (除非有无穷多个) 相交时刻。返回的相交时刻保证在绝对误差 `fuzz` 以内正确, 或者, 如果 `fuzz < 0`, 则达到机器精度。

```
real[] times(path p, real x)
```

<sup>18</sup>此算法可简单描述如下: 设  $p$  与  $q$  的长度分别为  $m$  和  $n$  (从而两路径分别有  $m$  段和  $n$  段), 先按字典序找出最小的相交的段 (即相交曲线段对中  $p$  中序号最小段和与之相交的  $q$  的段中序号最小的段), 设此段中某交点时刻  $(u, v)$  按二进制小数表示分别为  $(.t_1 t_2 \dots)_2$  和  $(.u_1 u_2 \dots)_2$ , 选取使得 “交错二进制” 表示  $(.t_1 u_1 t_2 u_2 \dots)_2$  最小的时刻对  $(u, v)$ 。不难看出, 这种算法的就是简单的一个二重循环找交点再进行排序的算法, 实现上较简易。——译者注

返回路径  $p$  与通过  $(x,0)$  点的竖直直线的所有相交时刻。

```
real[] times(path p, explicit pair z)
```

返回路径  $p$  与通过  $(0,z.y)$  点的水平直线的所有相交时刻。

```
real[] mintimes(path p)
```

返回长度为 2，包含路径  $p$  分别达到其最小水平与竖直范围的时刻的数组。

```
real[] maxtimes(path p)
```

返回长度为 2，包含路径  $p$  分别达到其最大水平与竖直范围的时刻的数组。

```
pair intersectionpoint(path p, path q, real fuzz=-1);
```

返回交点  $\text{point}(p, \text{intersect}(p, q, \text{fuzz})[0])$ 。

```
pair[] intersectionpoints(path p, path q, real fuzz=-1);
```

返回包含路径  $p$  与  $q$  所有交点的数组。

```
pair extension(pair P, pair Q, pair p, pair q);
```

返回线段  $P-Q$  与  $p-q$  延长线的交点，否则，如果两直线平行，返回  $(\text{infinity}, \text{infinity})$ 。

```
slice cut(path p, path knife, int n);
```

以结构体 `slice`：

```
struct slice {
    path before, after;
}
```

返回路径  $p$  与路径 `knife` 第  $n$  次相交前后的路径  $p$  的划分（如果未找到交点，整个路径将被认为是在相交之“前”）。参数  $n$  会与相交次数取模。

```
slice firstcut(path p, path knife);
```

等价于 `cut(p, knife, 0)`；注意 `firstcut.after` 起到 METAPOST `cutbefore` 命令的作用。

```
slice lastcut(path p, path knife);
```

等价于 `cut(p, knife, -1)`；注意 `lastcut.before` 起到 METAPOST `cutafter` 命令的作用。

```
path buildcycle(... path[] p);
```

依照 METAPOST `buildcycle` 命令的行为<sup>19</sup>, 返回围绕由两个或更多相交的一系列路径连续地界定的区域得到的路径。

```
pair min(path p);
```

返回路径 `p` 的路径边框的 (左, 下) 角复数。

```
pair max(path p);
```

返回路径 `p` 的路径边框的 (右, 上) 角复数。

```
int windingnumber(path p, pair z);
```

返回环形路径 `p` 关于点 `z` 的卷绕数<sup>20</sup>。若路径以逆时针方向环绕 `z`, 则卷绕数是正数。如果 `z` 在路径 `p` 上, 则返回常数 `undefined` (定义为最大的奇数)。

```
bool interior(int windingnumber, pen fillrule);
```

据填充规则 `fillrule`, 当 `windingnumber` 对应一个内点时, 返回 `true`<sup>21</sup>。

```
bool inside(path p, pair z, pen fillrule=currentpen);
```

返回 `true` 当且仅当点 `z` 在环形路径 `p` 据填充规则 `fillrule` (见 36 页 6.3 节) 界定的区域内部或边界上。

```
int inside(path p, path q, pen fillrule=currentpen);
```

如果环形路径 `p` 按填充规则 `fillrule` (见 36 页 6.3 节) 严格包含 `q` 则返回 1, 如果路径 `q` 严格包含 `p` 则返回 -1, 否则返回 0。

```
pair inside(path p, pen fillrule=currentpen);
```

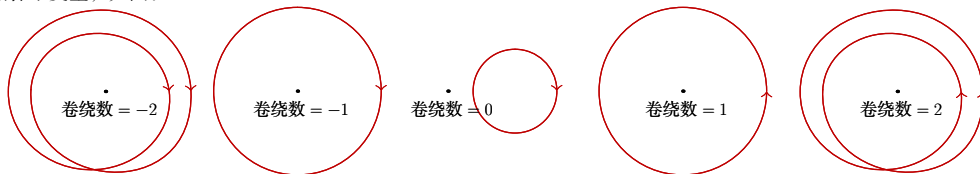
返回按填充规则 `fillrule` (见 36 页 6.3 节) 在一条环形路径 `p` 严格内部的任意一点。

```
path[] strokepath(path g, pen p=currentpen);
```

返回 PostScript 用画笔 `p` 绘制路径 `g` 时将会填充的路径数组。

<sup>19</sup>在 METAPOST 中的行为方式是: 对于路径  $p_1, p_2, \dots, p_k$ , 在每个  $p_i$  和  $p_{i+1}$  之间选取对  $p_i$  尽可能晚而对  $p_{i+1}$  尽可能早的交点, 将其沿路径连成封闭环形。参看 *METAPOST: A User's Manual* 中 9.1 节 Building Cycles。——译者注

<sup>20</sup>这是一个拓扑不变量, 如图:



用数学公式表示, 卷绕数  $= \tilde{\theta}(t_0) - \tilde{\theta}(0) = \frac{1}{2\pi i} \oint_C \frac{dz}{z-a}$ , 其中  $C: z = z(t), (t \in [0, t_0])$  为复平面上的闭曲线,  $z - a$  的极坐标表示为  $(\tilde{r}(t), \tilde{\theta}(t))$ 。——译者注

<sup>21</sup>这是一个简单的函数, 当 `fillrule` 为 `zerowinding` 时, 返回值等价于 `windingnumber != 0`; 当 `fillrule` 为 `evenodd` 时, 返回值等价于 `windingnumber%2 != 0` 即 `windingnumber` 是奇数。——译者注

`guide` 未解析的三次样条（一系列三次样条结点与控制点）。路向（`guide`）的隐式初始式为 `nullpath`；这对在循环中建立路向有用。

路向与路径类似，只是三次样条的计算被延迟到绘制时（当它解析为一条路径时）；这就允许具有自由终点情形的两条路向光滑地相连。下面例子中的实线是作为路向逐次建立的，但仅在绘制时解析；而虚线则在每次迭代时，在整个结点集（用红色表示）尚未都知道时就逐次解析：

```
size(200);

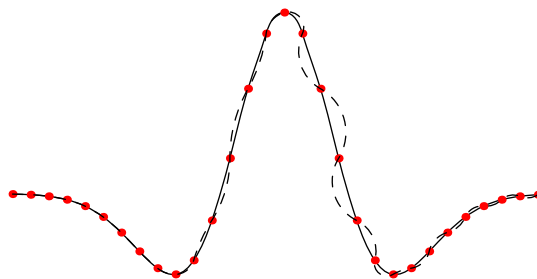
real mexican(real x) {return (1-8x^2)*exp(-(4x^2));}

int n=30;
real a=1.5;
real width=2a/n;

guide hat;
path solved;

for(int i=0; i < n; ++i) {
    real t=-a+i*width;
    pair z=(t,mexican(t));
    hat=hat..z;
    solved=solved..z;
}

draw(hat);
dot(hat,red);
draw(solved,dashed);
```



我们指出路向和路径的使用中的一个有效区别：

```
guide g;
for(int i=0; i < 10; ++i)
```

```
g=g--(i,i);
path p=g;
```

以线性时间运行，而

```
path p;
for(int i=0; i < 10; ++i)
    p=p--(i,i);
```

以平方时间运行，因为在每步迭代中整个路径直到那一点都被复制。

下列过程可用于在不将路向实际解析为确定的路径的情况下检查路向的个别元素（除了内部环形，它们会被解析）：

```
int size(guide g);
```

与 `size(path p)` 类似。

```
int length(guide g);
```

与 `length(path p)` 类似。

```
bool cyclic(path p);
```

与 `cyclic(path p)` 类似。

```
pair point(guide g, int t);
```

与 `point(path p, int t)` 类似。

```
guide reverse(guide g);
```

与 `reverse(path p)` 类似。如果 `g` 是环形的而且也包含次级环，它首先会被解析为一条路径，然后反向。如果 `g` 不是环形的但包含内部环，只有内部环先会被解决。如果没有内部环，路向会反向但不会解析为路径。

```
pair[] dirSpecifier(guide g, int i);
```

返回一个长度为 2 的复数数组，包含对路向 `g` 在结点 `i` 和 `i+1` 之间的段的出向（在元素 0）和入向（在元素 1）的方向限定子<sup>22</sup>（或 (0,0)，如果没有限定）。

```
pair[] controlSpecifier(guide g, int i);
```

如果路向 `g` 在结点 `i` 和 `i+1` 之间的段有明确的出向和入向控制点<sup>23</sup>，它们会被分别以一个两元素的数组的元素 0 和 1 返回。否则，会返回一个空白数组。

<sup>22</sup>在形如 `z1 {d1} .. {d2} z2` 的路向中，出向和入向的方向限定子分别为 `d1` 和 `d2`，模长会被约化为 1。——译者注

<sup>23</sup>在形如 `z1 .. controls c1 and c2 .. z2` 的路向中，出向和入向的控制点分别为 `c1` 和 `c2`。路向 `z1 .. controls c .. z2` 等价于 `z1 .. controls c and c .. z2`。——译者注

```
tensionSpecifier tensionSpecifier(guide g, int i);
```

返回路向 `g` 在结点 `i` 和 `i+1` 之间的段的张力限定子。`tensionSpecifier` 类型的个别组成部分可以按虚拟成员 `in`、`out` 和 `atLeast` 访问<sup>24</sup>。

```
real[] curlSpecifier(guide g);
```

返回一个数组，它包含路向 `g` 的起点卷曲限定子（在元素 0 中）和终点卷曲限定子（在元素 1 中）<sup>25</sup>。

作为技术细节我们指出，给 `nullpath` 的方向限定子将改变另一边的结点：路向

```
a..{up>nullpath..b;
c..nullpath{up}..d;
e..{up>nullpath{down}..f;
```

分别等价于

```
a..nullpath..{up}b;
c{up}..nullpath..d;
e{down}..nullpath..{up}f;
```

## 6.3 画笔

在 *Asymptote* 中，画笔为四个基本绘图命令提供了上下文（见 7 页第四章）。它们用于限定如下的绘图属性：颜色、线型、线宽、线端、线连接、填充规则、文字对齐、字体、字体大小、图案、覆写模式以及笔尖的书写变换。绘图过程默认使用的画笔叫做 `currentpen`。它提供了与 *METAPOST* 命令 `pickup` 相当的功能。画笔的隐式初始式为 `defaultpen`。

画笔可以用非结合的二元运算符 `+` 相加。这将加起两个画笔的颜色。右端画笔的所有其他非默认属性将覆盖左端的画笔。因而，比如说，可以用 `dashed+red+green` 或 `red+green+dashed` 或 `red+dashed+green` 来得到黄色虚线笔。二元运算符 `*` 可用于以一个实数放缩画笔的颜色，直至一个或多个颜色分量等于 1 而饱和<sup>26</sup>。

颜色用下列颜色空间之一限定：

```
pen gray(real g);
```

<sup>24</sup>在形如 `z1 .. tension t1 and t2 .. z2` 的路向中，成员 `in`、`out` 分别为实数 `t1`、`t2`，`atLeast` 为 `false`；在形如 `z1 .. tension atLeast t1 and t2 .. z2` 的路向中，`atLeast` 的值为 `true`。路向 `z1 .. tension t .. z2` 等价于 `z1 .. tension t and t .. z2`。对于 `tensionSpecifier` 类型的变量 `tens`，可以使用 `z1 .. tens .. z2` 的语法构造路向。——译者注

<sup>25</sup>在形如 `z1 {curl c1} .. {curl c2} z2` 的路向中，起、终点限定子分别为 `c1` 和 `c2`——译者注

<sup>26</sup>通常而言，使用加号和乘号调色并不直观而且易错（如导致颜色分量大于 1）。除了使用预定义颜色以及在颜色空间中直接计算各分量外，比较方便自然的方式是使用 6.9.1 节介绍的线性内插函数 `interp(T a, T b, real t)` 进行混色，而不必关心具体的分量（译者曾为此功能写过运算符重载的混色模块，语法接近 *L<sup>A</sup>T<sub>E</sub>X* 的 `xcolor` 宏包）。——译者注



产生灰阶颜色，其中亮度  $g$  在区间  $[0, 1]$  内，0.0 表示黑色而 1.0 表示白色。

```
pen rgb(real r, real g, real b);
```

产生 RGB 颜色<sup>27</sup>，其中红、绿、蓝每个的亮度都在区间  $[0, 1]$  内。

```
pen cmyk(real c, real m, real y, real k);
```

产生 CMYK 颜色<sup>28</sup>，其中青、品红、黄、黑每个的亮度都在区间  $[0, 1]$  内。

```
pen invisible;
```

这是使用不可见墨水书写的特殊画笔，但就像什么东西被画了一样调节边框（类似  $\text{T}_{\text{E}}\text{X}$  中的 `\phantom` 命令）。函数 `bool invisible(pen)` 可用来测试一支画笔是否不可见。

默认的颜色是 `black`（黑色）；这个可以使用过程 `defaultpen(pen)` 来改变。函数 `colorspace(pen p)` 以字符串返回画笔  $p$  的颜色空间（`"gray"`、`"rgb"`、`"cmyk"` 或 `" "`）。

函数 `real[] colors(pen)` 返回画笔的各颜色分量。函数 `pen gray(pen)`、`pen rgb(pen)` 以及 `pen cmyk(pen)` 返回分别将其参数转化为对应颜色空间的得到的新画笔。函数 `colorless(pen=currentpen)` 返回其参数的复本，而剥离其颜色属性（以避免混色）。

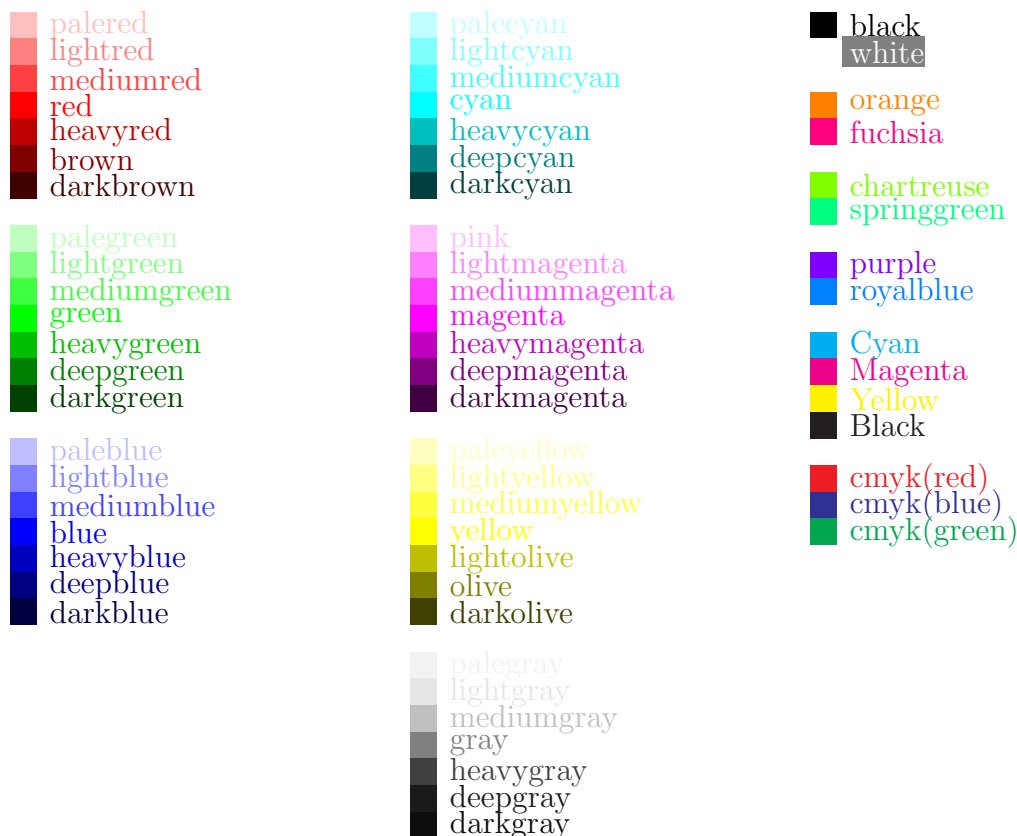
使用如下过程，可以将 6 字符的 RGB 十六进制串转换为画笔：

```
pen rgb(string s);
```

在 `plain` 模块中预定义了各种渐变与混合色的颜色名，它们基于灰阶基本色 `black`（黑）与 `white`（白），RGB 基本色 `red`（红）、`green`（绿）、`blue`（蓝），以及 RGB 次级基本色 `cyan`（青）、`magenta`（品红）、`yellow`（黄），还有 CMYK 基本色 `Cyan`、`Magenta`、`Yellow`、`Black`：

<sup>27</sup>红（red）、绿（green）、蓝（blue）颜色，这三种颜色是色光三原色，计算机屏幕输出即以这种颜色空间为基础。——译者注

<sup>28</sup>青（cyan）、品红（magenta）、黄（yellow）、黑（black）颜色，这是印刷工业常用的标准颜色空间。——译者注



标准的 140 种 RGB X11 颜色可以使用如下命令导入：

```
import x11colors;
```

而标准的 68 种 CMYK T<sub>E</sub>X 颜色可以使用如下命令导入：

```
import texcolors;
```

注意两个标准重复定义的一些颜色（如 `Green`）实际并不一致。

Asymptote 还附有一个 `asycolors.sty` 的 L<sup>A</sup>T<sub>E</sub>X 宏包，定义了 L<sup>A</sup>T<sub>E</sub>X CMYK 版本的 Asymptote 预定义颜色，从而使其可以直接用于 L<sup>A</sup>T<sub>E</sub>X 字符串中。通常，这些颜色通过画笔参数传递给 L<sup>A</sup>T<sub>E</sub>X；然而，要仅对字符串的一部分改变颜色，比如说在幻灯演示中（参看原手册 7.17 节 `slide` 部分<sup>29</sup>，可能需要直接向 L<sup>A</sup>T<sub>E</sub>X 设置颜色。这个文件可以用 Asymptote 命令传递给 L<sup>A</sup>T<sub>E</sub>X：

```
usepackage("asycolors");
```

定义于 `plain_pens.asy` 的结构体 `hsv` 可以用于在 HSV（Hue Saturation Value，色调饱和度明度）和 RGB 空间相互转换，其中色调 `h` 是一个在  $[0, 360)$  内的角度而饱和度 `s` 和明度 `v` 在  $[0, 1]$  内：

```
pen p=hsv(180,0.5,0.75);
```

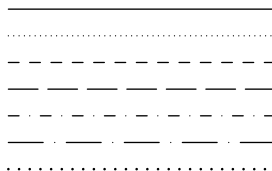
<sup>29</sup>本译本未包括此部分。——译者注

```
write(p);           // ([default], red=0.375, green=0.75, blue=0.75)
hsv q=p;
write(q.h,q.s,q.v); // 180    0.5    0.75
```

线型用函数 `pen linetype(string s, real offset=0, bool scale=true, bool adjust=true)` 确定，其中 `s` 是一个由空格分隔的若干整数或实数组成的字符串。可选参数 `offset` 限定在模式的何处开始。第一个数字限定在画笔落下时画多远（如果 `scale` 为 `true`，以画笔线宽为单位；否则使用 PostScript 单位），第二个数字限定在画笔抬起时画多远，以此类推。如果 `adjust` 为 `true`，这此间距将自动由 `Asymptote` 调整以适合路径的弧长。这里是一些预定义的线型：

```
pen solid=linetype("");
pen dotted=linetype("0 4");
pen dashed=linetype("8 8");
pen longdashed=linetype("24 8");
pen dashdotted=linetype("8 8 0 8");
pen longdashdotted=linetype("24 8 0 8");
pen Dotted=dotted+1.0;
pen Dotted(pen p=currentpen) {return dotted+2*linewidth(p);}

```



默认的线型是 `solid`；这可以用 `defaultpen(pen)` 改变。

画笔的线宽使用 `pen linewidth(real)` 按 PostScript 单位来限定<sup>30</sup>。默认的线宽为 0.5 dp；这个值可以用 `defaultpen(pen)` 来修改。画笔的线宽可由 `real linewidth(pen p=currentpen)` 返回。为方便计，在模块 `plain` 中我们定义了

```
static void defaultpen(real w) {defaultpen(linewidth(w));}
static pen operator +(pen p, real w) {return p+linewidth(w);}
static pen operator +(real w, pen p) {return linewidth(w)+p;}

```

从而可以像这样设置线宽：

```
defaultpen(2);
pen p=red+0.5;
```

特定 PostScript 线端的画笔可以通过以一个整数参数调用 `linecap` 返回<sup>31</sup>：

<sup>30</sup>在 `Asymptote` 中，用 PostScript 单位表示的长度是最终输出图形中的绝对长度，不会被放缩。——译者注  
<sup>31</sup>线端（line cap）指曲线端点的形状，PostScript 三种线端有方形、圆形和外扩方形，如下图所示：

```
pen squarecap=linecap(0);
pen roundcap=linecap(1);
pen extendcap=linecap(2);
```

默认的线端为 `roundcap`，可以用 `defaultpen(pen)` 改变。画笔的线端由 `int linecap(pen p=currentpen)` 返回。

特定 PostScript 连接样式的画笔可以以一个整数参数调用 `linejoin` 返回<sup>32</sup>：

```
pen miterjoin=linejoin(0);
pen roundjoin=linejoin(1);
pen beveljoin=linejoin(2);
```

默认的连接样式为 `roundjoin`，可以用 `defaultpen(pen)` 改变。画笔的连接样式由 `int linejoin(pen p=currentpen)` 返回。

有一定 PostScript 尖角限量（miter limit）的画笔由 `miterlimit(real)` 的调用返回。默认的尖角限量，10.0，可以用 `defaultpen(pen)` 改变。画笔的尖角限量由 `real miterlimit(pen p=currentpen)` 返回。




特定 PostScript 填充规则的画笔可以以一个整数参数调用 `fillrule` 返回：

```
pen zerowinding=fillrule(0);
pen evenodd=fillrule(1);
```

填充规则确定如何判定是否在一个路径或路径数组内部的算法，它仅影响 `clip`、`fill` 和 `inside` 函数。对于 `zerowinding` 填充规则，如果路径与水平线 `z--z+infinity` 向上的相交次数减去向下的相交次数等于零，则点 `z` 在路径限界的区域内部。对 `evenodd` 填充规则，如果上述相交的总数为偶数，则认为 `z` 在区域外面。默认的填充规则为 `zerowinding`，可以用 `defaultpen(pen)` 改变。画笔的填充规则由 `int fillrule(pen p=currentpen)` 返回。

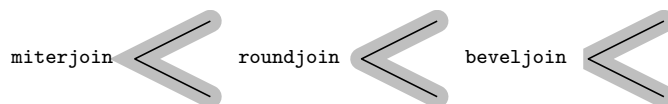
特定文字对齐设置的画笔能以一个整数参数调用 `basealign` 返回：

```
pen nobasealign=basealign(0);
pen basealign=basealign(1);
```

squarecap   
 roundcap   
 extendcap 

——译者注

<sup>32</sup>PostScript 三种折线的连接样式为尖角连接、圆角连接和斜角连接：



——译者注

默认设置为 `nobasealign`，会令标签对齐过程使用整个标签的边框进行对齐，它可用 `defaultpen(pen)` 改变。相反，`baseline` 要求考虑  $\text{T}_{\text{E}}\text{X}$  基线。画笔的基线对齐设置由 `int basealign(pen p=currentpen)` 返回。

字体大小用函数 `pen fontsize(real size, real lineskip=1.2*size)` 按  $\text{T}_{\text{E}}\text{X}$  点 ( $1\text{pt} = 1/72.27$  英寸) 确定。默认的字体大小为  $12\text{pt}$ ，可以用 `defaultpen(pen)` 改变。非标准的字体大小需要在文件开头插入

```
import fontsize;
```

(这需要 `fix-cm` 宏包，它可从 <http://www.ctan.org/tex-archive/help/Catalogue/entries/fix-cm> 获得，也包含在最新的  $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$  发行版中)。画笔的字体大小与行间距可以分别用过程 `real fontsize(pen p=currentpen)` 与 `real lineskip(pen p=currentpen)` 进行检查。

使用特定  $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$  NFSS 字体的画笔可调用函数

```
pen font(string encoding, string family, string series, string shape)
```

返回。默认设置为 `font("OT1","cmr","m","n")`，对应  $12\text{pt}$  Computer Modern Roman 字体；它可用 `defaultpen(pen)` 改变。画笔的字体设置由 `string font(pen p=currentpen)` 返回。国际标准化字符集支持由 `unicode` 包提供（见原手册 7.19 节 `unicode` 部分<sup>33</sup>）。

另外，可以使用函数 `pen font(string name)` 选择固定大小的  $\text{T}_{\text{E}}\text{X}$  字体（此时 `fontsize` 不起作用），如 `"cmr12"`（ $12\text{pt}$  Computer Modern Roman）或 `pcrr`（Courier）。还可以给定可选的大小参数来将字体放缩到要求的大小：`pen font(string name, real size)`。

非标准的字体命令可以用 `pen fontcommand(string)` 生成。

下面提供了标准 PostScript 字体的方便界面：

```
pen AvantGarde(string series="m", string shape="n");
pen Bookman(string series="m", string shape="n");
pen Courier(string series="m", string shape="n");
pen Helvetica(string series="m", string shape="n");
pen NewCenturySchoolBook(string series="m", string shape="n");
pen Palatino(string series="m", string shape="n");
pen TimesRoman(string series="m", string shape="n");
pen ZapfChancery(string series="m", string shape="n");
pen Symbol(string series="m", string shape="n");
pen ZapfDingbats(string series="m", string shape="n");
```

画笔的透明度可以用此命令改变：

```
pen opacity(real opacity=1, string blend="Compatible");
```

<sup>33</sup> 此译本未包括此部分。——译者注

不透明度 (opacity) 可以从 0 (完全透明) 到默认值 1 (不透明) 变动, 而 `blend` 确定下列前景-背景混合操作之一:

```
"Compatible", "Normal", "Multiply", "Screen", "Overlay", "SoftLight",
"HardLight", "ColorDodge", "ColorBurn", "Darken", "Lighten", "Difference",
"Exclusion", "Hue", "Saturation", "Color", "Luminosity"
```

它们在 <http://partners.adobe.com/public/developer/en/pdf/PDFReference16.pdf> 中描述。

由于 PostScript 不支持透明, 这个特性仅对 `-f pdf` 输出格式选项有效。可以从 PDF 结果文件通过 ImageMagick `convert` 程序生成。标签永远以 `opacity` 值 1 绘制。透明填充的一个简单例子在示例文件 `transparency.asy` 中给出<sup>34</sup>。

`picture` 中的 PostScript 命令可以用于创建一个以字符串 `name` 标识的铺砌图案, 通过将其加在全局 PostScript 帧 `currentpatterns` 上来影响 `fill` 和 `draw` 操作, 外带可选的左下边界 `lb` 和右上边界 `rt`。

```
import patterns;
void add(string name, picture pic, pair lb=0, pair rt=0);
```

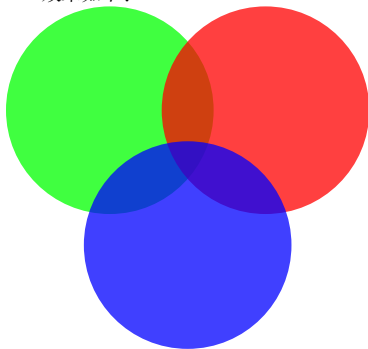
要使用图案 `name` 进行 `fill` 或 `draw`, 使用画笔 `pattern("name")`。例如, 矩形铺砌可以使用定义于 `patternss.asy` 的过程

```
picture tile(real Hx=5mm, real Hy=0, pen p=currentpen,
             filltype filltype=NoFill);
picture checker(real Hx=5mm, real Hy=0, pen p=currentpen);
picture brick(real Hx=5mm, real Hy=0, pen p=currentpen);
```

来构造:

```
size(0,90);
import patterns;
```

<sup>34</sup>效果如下:



```
size(0,150);

if(settings.outformat == "")
    settings.outformat="pdf";

begingroup();
fill(shift(1.5dir(120))*unitcircle,green+opacity(0.75));
fill(shift(1.5dir(60))*unitcircle,red+opacity(0.75));
fill(unitcircle,blue+opacity(0.75));
endgroup();
```

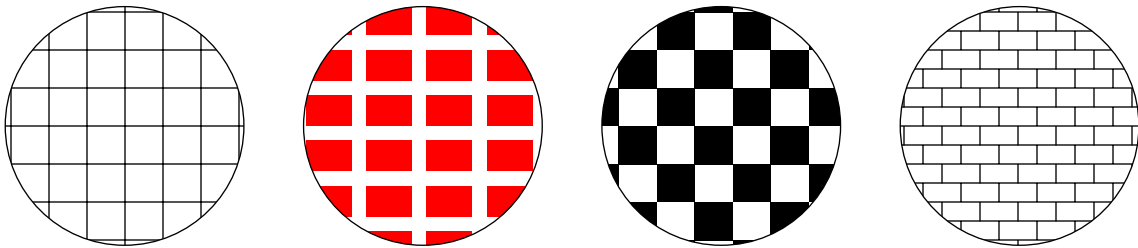
——译者注

```

add("tile",tile());
add("filledtilewithmargin",tile(6mm,4mm,red,Fill),(1mm,1mm),(1mm,1mm));
add("checker",checker());
add("brick",brick());

real s=2.5;
filldraw(unitcircle,pattern("tile"));
filldraw(shift(s,0)*unitcircle,pattern("filledtilewithmargin"));
filldraw(shift(2s,0)*unitcircle,pattern("checker"));
filldraw(shift(3s,0)*unitcircle,pattern("brick"));

```



阴影线图案可以通过过程 `picture hatch(real H=5mm, pair dir=NE, pen p=currentpen)`、`picture crosshatch(real H=5mm, pen p=currentpen)` 生成：

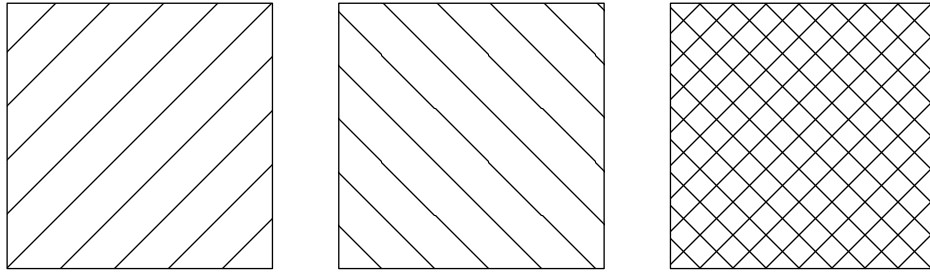
```

size(0,100);
import patterns;

add("hatch",hatch());
add("hatchback",hatch(NW));
add("crosshatch",crosshatch(3mm));

real s=1.25;
filldraw(unitsquare,pattern("hatch"));
filldraw(shift(s,0)*unitsquare,pattern("hatchback"));
filldraw(shift(2s,0)*unitsquare,pattern("crosshatch"));

```

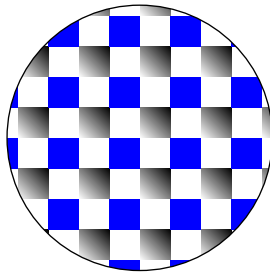


你可能需要在你的 PostScript 预览器中关闭锯齿以正确显示。自定义图案可以按照 `patterns.asy` 的例子简单地构造。铺砌的图案甚至可以包含渐变（见 9 页 4.2 关于渐变填充的部分），下面是示例<sup>35</sup>：

```
size(0,100);
import patterns;

real d=4mm;
picture tiling;
path square=scale(d)*unitsquare;
axialshade(tiling,square,white,(0,0),black,(d,d));
fill(tiling,shift(d,d)*square,blue);
add("shadedtiling",tiling);

filldraw(unitcircle,pattern("shadedtiling"));
```



可以使用 `pen makepen(path)` 来以任意多边形路径确定自定义的笔尖；此路径表示用以绘制包含单个点的路径的记号。笔尖路径可以用 `path nib(pen)` 从一个画笔中恢复出来。不像 METAPOST 中那样，路径不必是凸的：

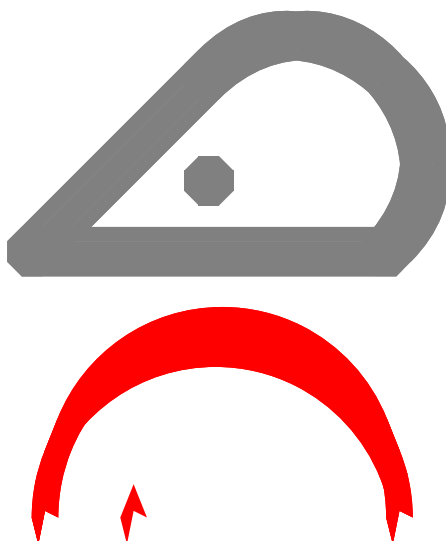
```
size(200);
pen convex=makepen(scale(10)*polygon(8))+grey;
draw((1,0.4),convex);
```

<sup>35</sup>原手册中有说明：“因为并非所有打印机支持 PostScript 3，（示例图形）不包括在手册中。”现在译本把它加进正文。——译者注



```
draw((0,0)---(1,1)..(2,0)--cycle,convex);

pen nonconvex=scale(10)*
  makepen((0,0)--(0.25,-1)--(0.5,0.25)--(1,0)--(0.5,1.25)--cycle)+red;
draw((0.5,-1.5),nonconvex);
draw((0,-1.5)..(1,-0.5)..(2,-1.5),nonconvex);
```



值 `nullpath` 表示一个圆形笔尖（默认值）；椭圆形画笔可以仅仅通过给画笔乘以一个变换得到：  
`yscale(2)*currentpen`。

使用画笔属性 `overwrite`，可以避免标签互相覆写，它取单独一个参数：

**Allow**

允许标签互相覆写。这是默认行为（除非用 `defaultpen(pen)` 废除）。

**Suppress**

禁止，对每个将覆写另一个标签的标签，连带一个警告。

**SuppressQuiet**

禁止，对每个将覆写另一个标签的标签，没有警告。

**Move**

把将要覆写另一个标签的标签移走并发出警告。由于此调整是在最终输出阶段（以 PostScript 坐标），可能结果会得到比要求的更大一些的图形。

**MoveQuiet**

把将要覆写另一个标签的标签移走而不发出警告。由于此调整是在最终输出阶段（以 PostScript 坐标），可能结果会得到比要求的更大一些的图形。

过程 `defaultpen()` 返回当前默认画笔属性。调用过程 `resetdefaultpen()` 可重设所有画笔默认属性为初始值。

## 6.4 变换

Asymptote 广泛利用仿射变换。复数  $(x, y)$  由变换  $t=(t.x, t.y, t.xx, t.xy, t.yx, t.yy)$  变换到  $(x', y')$ ，其中

$$\begin{aligned} x' &= t.x + t.xx * x + t.xy * y \\ y' &= t.y + t.yx * x + t.yy * y \end{aligned}$$

这等价于 PostScript 变换  $[t.xx \ t.yx \ t.xy \ t.yy \ t.x \ t.y]$ 。

变换可以用左乘（通过二元运算符  $*$ ）应用于复数、路向、路径、画笔、字符串、变换、帧与图（见 27 页 6.2 节圆的例子）。变换可以互相复合并通过函数 `transform inverse(transform t)` 取逆；它们也可以用  $\wedge$  运算符进行任意整数次幂<sup>36</sup>。

内建的变换有：

```
transform identity();
```

恒同变换；

```
transform shift(pair z);
```

按复数  $z$  平移；

```
transform shift(real x, real y);
```

按复数  $(x, y)$  平移；

```
transform xscale(real x);
```

在  $x$  方向按  $x$  放缩；

```
transform yscale(real y);
```

在  $y$  方向按  $y$  放缩；

```
transform scale(real s);
```

在  $x$  与  $y$  方向都按  $s$  放缩；

```
transform scale(real x, real y);
```

在  $x$  方向按  $x$  而在  $y$  方向按  $y$  放缩；

```
transform slant(real s);
```

<sup>36</sup>包括负整数和零。变换的  $n$  次幂 ( $n > 0$ ) 表示  $n$  个自身复合；变换的  $-n$  次幂表示其逆变换的  $n$  次幂；变换的零次幂就是恒同变换。——译者注

映射  $(x, y) \mapsto (x + s \cdot y, y)$ ;

```
transform rotate(real angle, pair z=(0,0));
```

关于  $z$  旋转 `angle` 度;

```
transform reflect(pair a, pair b);
```

关于直线  $a--b$  反射。

变换的隐式初始式是 `identity()`。过程 `shift(transform t)` 和 `shiftless(transform t)` 分别返回变换  $(t.x, t.y, 0, 0, 0, 0)$  和  $(0, 0, t.xx, t.xy, t.yx, t.yy)$ <sup>37</sup>。

## 6.5 帧 (frame) 与图 (picture)

`frame` 帧 (`frame`) 是在 PostScript 坐标中绘画的画布。偶尔需要为了构造延时作图过程而直接在帧上工作，通常在图 (`picture`) 上工作更为方便。帧的隐式初始式为 `newframe`。仅当帧 `f` 是空的时，函数 `bool empty(frame f)` 返回 `true`。一帧可以用 `erase(frame)` 过程擦除。函数 `pair min(frame)` 和 `pair max(frame)` 分别返回帧边框的 (左, 下) 和 (右, 上) 坐标。用命令

```
void add(frame dest, frame src);
```

可以将帧 `src` 的内容附加在帧 `dest` 的后面，或用命令

```
void prepend(frame dest, frame src);
```

添加在前面<sup>38</sup>。

按照与 `label` 使用 `align` 参数 (参看 12 页 4.4 节标签部分) 类似的方式，把帧 `f` 按方向 `align` 对齐得到的帧，可由

```
frame align(frame f, pair align);
```

返回。

为描画或填充围绕标签或帧的盒形或椭圆并返回边界路径，使用如下预定义的 `envelope` 过程之——<sup>39</sup>：

```
path box(frame f, Label L="", real xmargin=0,
          real ymargin=xmargin, pen p=currentpen,
```

<sup>37</sup>变换除了可以相乘、取幂，也可以相加，变换相加就是对应平移向量和矩阵相加。

`Asymptote` 并没有提供直接控制变换各分量的方式，可以用名字访问 `xx`、`xy` 等成员，但它们是只读的。不过，通过变换的运算，可以使用表达式

```
shift(x,y) * (rotate(90)*scale(yx,-xy) + scale(xx,yy))
// (x, y, xx, xy, yx, yy)
```

直接以矩阵元素值来表示任意的仿射变换。——译者注

<sup>38</sup>附加在后面的帧在较上的位置 (较晚绘制)，添加在前面的帧在较下的位置 (较早绘制)。——译者注

<sup>39</sup>这几个过程使用对应的画笔、填充类型等在帧 `f` 中绘制对应的形状 (其边界由 `f` 原有的内容和标签 `L` 决定)，并返回此形状的边界路径。——译者注

```

        filltype filltype=NoFill, bool above=true);
path roundbox(frame f, Label L="", real xmargin=0,
              real ymargin=xmargin, pen p=currentpen,
              filltype filltype=NoFill, bool above=true);
path ellipse(frame f, Label L="", real xmargin=0,
             real ymargin=xmargin, pen p=currentpen,
             filltype filltype=NoFill, bool above=true);

```

`picture` 图是在模块 `plain` 中定义的高级结构（参见 58 页 6.8 节结构体部分），提供以用户坐标绘图的画布。默认的图叫做 `currentpicture`。新的图可以这样创建：

```
picture pic;
```

匿名图可以用表达式 `new picture` 创建。

`size` 过程限定要求的图的尺寸：

```

void size(picture pic=currentpicture, real x, real y=x,
          bool keepAspect=Aspect);

```

如果  $x$  与  $y$  的大小都为 0，用户坐标将被解释为 PostScript 坐标。在这种情况下，把 `pic` 映射到最终的输出帧的变换为 `identity()`。

如果  $x$  或  $y$  恰有一个为 0，该方向则不施加限制；它将与另一方向一致地放缩。

如果 `keepAspect` 设为 `Aspect` 或 `true`，该图将按原比例放缩，并保持最终宽度不大于  $x$  而最终高度不大于  $y$ 。

如果 `keepAspect` 设为 `IgnoreAspect` 或 `false`，该图将以两个方向放缩以使最终宽度为  $x$  而最终高度为  $y$ 。

要令图 `pic` 的用户坐标表示在  $x$  方向的  $x$  单位倍和  $y$  方向的  $y$  单位倍，使用

```
void unitsize(picture pic=currentpicture, real x, real y=x);
```

如果非零，这里  $x$  与  $y$  值将覆盖图 `pic` 的对应尺寸参数。

过程

```

void size(picture pic=currentpicture, real xsize, real ysize,
          pair min, pair max);

```

强迫最终的图放缩为把用户坐标 `box(min,max)` 映射到宽 `xsize` 高 `ysize` 的区域（当这些参数非零时）。

此外，调用过程

```

transform fixedscaling(picture pic=currentpicture, pair min,
                       pair max, pen p=nullpen, bool warn=false);

```

将使图 `pic` 使用一个固定的放缩把 `box(min,max)` 中的用户坐标映射为 (已经限定的) 图尺寸, 其中考虑画笔 `p` 的宽度。如果最终的图超出限定的尺寸将发出警告。

调用函数 `shipout`, 可以将图 `pic` 装入帧, 并使用图像格式 `format` 输出到文件 `prefix.fotmat`:

```
void shipout(string prefix=defaultfilename, picture pic=currentpicture,
             orientation orientation=orientation,
             string format="", bool wait=false, bool view=true,
             string options="", string script="",
             projection P=currentprojection);
```

默认的输出格式为 PostScript, 可用命令行选项 `-f` 或 `-tex` 改变。参数 `options`, `script` 与 `projection` 仅与 3D 图相关。如果 `defaultfilename` 为空串, 则将使用前缀 `outprefix()`。

如果之前没有执行过 `shipout()` 命令, 在文件末尾将会隐式地加上一条 `shipout()` 命令。

默认的页面方向为 `Portrait` (纵向); 这可以通过改变变量 `orientation` 修改。要以横向模式输出, 仅仅需要设置 `orientation=Landscape` 或发出指令

```
shipout(Landscape);
```

要旋转页面  $-90^\circ$ , 使用方向 `Seascape`。要旋转页面  $180^\circ$ , 使用方向 `UpsideDown` (上下颠倒)。

可以调用函数明确将图 `pic` 装入一帧:

```
frame pic.fit(real xsize=pic.xsize, real ysize=pic.ysize,
              bool keepAspect=pic.keepAspect);
```

默认的尺寸和宽高比设置是给 `size` 命令的值 (默认分别为 0, 0 和 `true`)。当前将用于把图 `pic` 装入一帧的变换可通过成员函数 `pic.calculateTransform()` 返回。

在某些只能对 `pic` 的近似尺寸做出估计的情形 (例如 2D 图表), 图装入过程

```
frame pic.scale(real xsize=this.xsize, real ysize=this.ysize,
                bool keepAspect=this.keepAspect);
```

(它将放缩结果帧, 包括标注和固定尺寸的对象) 将强制完全遵从给定的尺寸设定, 但通常应该并不需要。

要在图外一定边距画出边框, 使用函数

```
frame bbox(picture pic=currentpicture, real xmargin=0,
            real ymargin=xmargin, pen p=currentpen,
            filltype filltype=NoFill);
```

将图装入帧。这里 `filltype` 限定为如下填充类型之一:

**FillDraw**

填充内部并绘出边界。

```
FillDraw(real xmargin=0, real ymargin=xmargin, pen fillpen=nullpen,
         pen drawpen=nullpen);
```

如果 fillpen 为 nullpen，则用正在绘图的画笔填充；否则用画笔 fillpen 填充；如果 drawpen 为 nullpen，则用 fillpen 填充；否则用 drawpen 填充。可以设定可选的边距 xmargin 和 ymargin。

**Fill**

填充内部。

```
Fill(real xmargin=0, real ymargin=xmargin, pen p=nullpen)
```

如果 p 为 nullpen，则用正在绘图的画笔填充；否则用画笔 p 填充。可以设定可选的边距 xmargin 和 ymargin。

**NoFill**

不填充。

**Draw**

仅绘制边界。

```
Draw(real xmargin=0, real ymargin=xmargin, pen p=nullpen)
```

如果 p 是 nullpen，用正在绘图的画笔绘制边界；否则用画笔 p 绘制。可以设定可选的边距 xmargin 和 ymargin。

**UnFill**

剪裁区域<sup>40</sup>。

```
UnFill(real xmargin=0, real ymargin=xmargin)
```

以四周边距 xmargin 和 ymargin 剪裁区域。

```
RadialShade(pen penc, pen penr)
```

从边框中心的 penc 到边界的 penr 放射状渐变填充。

例如，要为一图画出有 0.25 cm 边距的边框并输出结果的帧，使用命令：

```
shipout(bbox(0.25cm));
```

<sup>40</sup>与 clip 命令的结果不同，UnFill 选项把区域内部剪掉，保留外部，而 clip 命令则保留被剪裁路径的内部。——译者注

用函数 `bbox(p, Fill)` 可以用画笔 `p` 的背景色将图装入帧。

函数

```
pair min(picture pic, user=false);
pair max(picture pic, user=false);
pair size(picture pic, user=false);
```

计算出如果图 `pic` 现在使用其默认尺寸设定装入一帧，它将具有的 PostScript 边界。如果 `user` 为 `false`，则返回值为 PostScript 坐标，否则以用户坐标返回。

函数

```
pair point(picture pic=currentpicture, pair dir, bool user=true);
```

是一种方便的方式确定相对图 `pic` 中心按方向 `dir` 得到的边框上的点，忽略固定尺寸对象（如标签和箭头）的贡献。如果 `user` 为 `false`，则返回值为 PostScript 坐标，否则以用户坐标返回。

函数

```
pair truepoint(picture pic=currentpicture, pair dir, bool user=true);
```

与 `point` 相同，只是它也计入固定尺寸的对象，使用如果图 `pic` 现在用其默认尺寸设定装入一帧将会具有的放缩变换。如果 `user` 为 `false`，则返回值为 PostScript 坐标，否则以用户坐标返回。

有时在分开的图上绘制对象使用 `add` 函数将图互相加入是有用的：

```
void add(picture src, bool group=true,
         filltype filltype=NoFill, bool above=true);
void add(picture dest, picture src, bool group=true,
         filltype filltype=NoFill, bool above=true);
```

第一个例子把 `src` 加入 `currentpicture`；第二个把 `src` 加入 `dest`。`group` 选项设定是否在图形用户界面 `xasy` 中要把 `src` 所有的元素视为单个实体（见原手册第 10 章 GUI<sup>41</sup>），`filltype` 要求可选的背景填充或剪裁，而 `above` 设定是否把 `src` 加到已有对象的上面还是下面。

同样还有过程将一个以 PostScript 坐标限定的图或帧 `src`，按照用户坐标 `position` 加入到另一图 `dest`（或 `currentpicture`）中。

```
void add(picture src, pair position, bool group=true,
         filltype filltype=NoFill, bool above=true);
void add(picture dest, picture src, pair position,
         bool group=true, filltype filltype=NoFill, bool above=true);
void add(picture dest=currentpicture, frame src, pair position=0,
         bool group=true, filltype filltype=NoFill, bool above=true);
void add(picture dest=currentpicture, frame src, pair position,
```

<sup>41</sup> 本译本未包括此章。——译者注

```
pair align, bool group=true, filltype filltype=NoFill,
bool above=true);
```

最后一种形式<sup>42</sup>中的可选参数 `align` 确定用于对齐帧的方向，行为与 `label` 的参数 `align` 类似（参看 12 页 4.4 节标签）。然而，一个关键区别是当 `align` 未设定时，标签是居中的，而帧与图则以 `position` 为原点对齐。帧对齐的例子可以在原手册 7.25 节 `graph` 模块的 `errorbars` 和 `image` 例子<sup>43</sup>。如果想对齐三个或更多子图，每次将它们分为两组：

```
picture pic1;
real size=50;
size(pic1,size);
fill(pic1,(0,0)--(50,100)--(100,0)--cycle,red);

picture pic2;
size(pic2,size);
fill(pic2,unitcircle,green);

picture pic3;
size(pic3,size);
fill(pic3,unitsquare,blue);

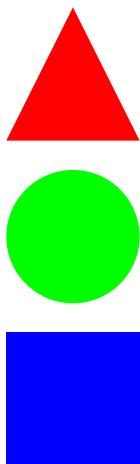
picture pic;
add(pic,pic1.fit(),(0,0),N);
add(pic,pic2.fit(),(0,0),10S);

add(pic.pic.fit(),(0,0),N);
add(pic.pic3.fit(),(0,0),10S);
```

<sup>42</sup>原文误作三种。文档这一部分多少与实际源代码不符合的，且为了易读性，有一些函数形式没有文档，有时令人相当莫名其妙。请读者在实际中遇到问题时自行查看 `plain_picture` 中的代码。——译者注

<sup>43</sup>本译本未包括此节。——译者注





此外，也可以使用 `attach` 自动增加图 `dest` 的尺寸以适应按用户坐标 `position` 加入的帧 `src`：

```
void attach(picture dest=currentpicture, frame src,
            pair position=0, bool group=true,
            filltype filltype=NoFill, bool above=true);
void attach(picture dest=currentpicture, frame src,
            pair position, pair align, bool group=true,
            filltype filltype=NoFill, bool above=true);
```

要清除一幅图的内容（但不清除尺寸设定），使用函数

```
void erase(picture pic=currentpicture);
```

要保存 `currentpicture`、`currentpen` 和 `currentprojection` 的快照，使用函数 `save()`。

要恢复 `currentpicture`、`currentpen` 和 `currentprojection` 的快照，使用函数 `restore()`。<sup>44</sup>

许多关于图和帧的进一步的操作由基本模块 `plain` 提供<sup>45</sup>。

使用如下过程之一，可以向一幅图中逐字插入 PostScript 命令：

```
void postscript(picture pic=currentpicture, string s);
void postscript(picture pic=currentpicture, string s, pair min,
                pair max)
```

这里 `min` 与 `max` 可以用于明确指定相关 PostScript 代码导致的边界。

逐字的 TeX 命令可以使用如下函数之一插入到中间 L<sup>A</sup>T<sub>E</sub>X 输出文件中：

```
void tex(picture pic=currentpicture, string s);
void tex(picture pic=currentpicture, string s, pair min, pair max);
```

<sup>44</sup> `save()` 函数和 `restore` 函数维护一个栈，可以嵌套使用。——译者注

<sup>45</sup> 然而手册中并没有给出更多的说明，因而可能需要用户自己阅读 `plain` 模块的相关代码。——译者注

这里 `min` 与 `max` 可以用于明确指定相关  $\text{T}_\text{E}\text{X}$  代码导致的边界。

要在  $\text{T}_\text{E}\text{X}$  导言区（在顶层模块的剩余部分有效<sup>46</sup>）发出一个全局的  $\text{T}_\text{E}\text{X}$  命令（如一个  $\text{T}_\text{E}\text{X}$  宏定义），使用：

```
void texpreamble(string s);
```

可以重设  $\text{T}_\text{E}\text{X}$  环境为初始状态，清除所有的宏定义，使用函数<sup>47</sup>

```
void texreset();
```

过程

```
void usepackage(string s, string options="");
```

提供了

```
texpreamble("\usepackage["+options+"]{"+s+"}");
```

的一个方便的缩写形式，可以用于导入  $\text{L}^{\text{A}}\text{T}_\text{E}\text{X}$  宏包。

## 6.6 文件

Asymptote 能读写文本文件（包括逗号分隔值文件 CSV）和可移植 XDR（External Data Representation 外部数据表示）二进制文件。

一个输入文件必须首先由 `input(string name, bool check=true, string comment="#")` 打开；于是读文件由赋值完成：

```
file fin=input("test.txt");
real a=fin;
```

如果可选的布尔参数 `check` 为 `false`，则不会检查文件是否存在。如果文件不存在或不可读，函数 `bool error(file)` 将返回 `true`。字符串 `comment` 的第一个字符指定一个注释字符。如果这个字符在数据文件中出现，此行的剩余部分会被忽略。当读字符串时，一个注释字符后紧跟另一个注释字符会被看作一个注释字符的字面量。

如果 `-globalwrite`（或 `-nosafe`）选项可用，则可以利用函数 `string cd(string s)` 改变当前工作目录为串 `s`，此函数返回新的工作目录。如果 `string s` 为空，路径将被重设为程序启动时的值。

当读入复数（序对）时，括号对是可选的。字符串也通过赋值读入，一起读入字符直到但不包括换行符。此外，Asymptote 提供函数 `string getc(file)` 读入下一个字符（它把注释字符当做普通字符）并将其作为一个字符串返回。

一个名为 `name` 的文件可以用

```
file output(string name, bool update=false);
```

<sup>46</sup>意谓在导言区的命令是全局有效的，能影响所有的标签，包括此函数执行前的标签。——译者注

<sup>47</sup>此函数的作用也是全局的，可以影响执行此函数之前的标签。——译者注

打开用作输出文件。如果 `update=false`，则文件中已有的全部数据将被清除，并且只能对文件做写操作。如果 `update=true`，则所有已有的数据都会被保留，读写位置会设为文件末尾，此时读写操作都可以使用。出于安全原因，只有在 `-globalwrite`（或 `-nosafe`）命令行选项被设置时，才允许向当前目录以外的文件中写入。

有两个特殊的文件：`stdin`（标准输入），它从键盘读入；以及 `stdout`（标准输出），它向终端输出。文件的隐式初始式为 `null`。

内部类型 `T` 的数据可以通过调用下列函数写入输出文件：

```
write(string s="", T x, suffix suffix=endl ... T[]);
write(file file, string s="", T x, suffix suffix=none ... T[]);
write(file file=stdout, string s="", explicit T[] x ... T[] []);
write(file file=stdout, T[] []);
write(file file=stdout, T[] [] []);
write(suffix suffix=endl);
write(file file, suffix suffix=none);
```

如果 `file` 未被指定，则 `stdout` 将被使用并且默认用换行符结束。如果指定，则可选的标识字符串 `s` 则会在数据 `x` 之前写入。在写入一维数组时，可以列出任意数量的数据值。参数 `suffix` 可以是下面几个中的一个：`none`（什么也不做），`flush`（刷新：立即输出缓存中的数据），`endl`（用换行符结束并刷新），`newl`（用换行符结束），`tab`（用制表位结束）或是 `comma`（用逗号结束）。这里有一些数据输出的简单例子：

```
file fout=output("test.txt");
write(fout,1);           // 写入 "1"
write(fout);             // 写入新的一行
write(fout,"List: ",1,2,3); // 写入 "List: 1 2 3"
```

一个文件也可以用 `xinput` 或 `xoutput` 而不是 `input` 或 `output` 打开，用来以 Sun Microsystem 公司的 XDR（External Data Representation 外部数据表示）可移植二进制格式（它在所有 UNIX 平台都可用）读写双精度（64 位）实数值和单精度（32 位）整数值。一个文件同样可以由 `bininput` 或 `boutput` 打开用来读写在本地机器的（不可移植）二进制格式双精度值。函数 `file single(file,0.0)` 可被用来设置一个文件读入单精度实数 XDR 或二进制值；调用 `file single(file,0.0,false)` 又将其设回读入双精度值。函数 `file single(file,0)` 和 `file single(file,0,false)` 可用于改变默认的整数精度（为单精度）。函数 `file single(file)` 和 `file single(file,false)` 可用于同时设置实数和整数二者的精度。

可以用布尔值函数 `eof(file)` 测试一个文件的末尾，用 `eol(file)` 测试行末尾，而用 `error(file)` 测试 I/O 错误。可以用 `flush(file)` 函数刷新输出缓存，用 `clear(file)` 清除原有的 I/O 错误，而用 `close(file)` 关闭文件。给定 `digits` 非零，函数 `int precision(file file=stdout, int digits=0)` 设置文件 `file` 的输出精度为 `digits` 位数，并返回原来的精度设置。函数 `int tell(file)` 返回一个文件相对开头的相对位置。过程 `seek(file file, int pos)` 可用来改变这个位置，而负的位置 `pos`

将被解释为相对文件末尾的位置。例如，可以用命令 `seek(file,0)` 倒回文件开头，用 `seek(file,-1)` 到达文件的最后一个字符处。命令 `seekeof(file)` 设置位置为文件的末尾。

设定 `settings.scroll=n` 为一个正整数 `n` 会要求每向 `stdout` 输出 `n` 行就停顿一下。而后可以按下回车键继续下面的 `n` 行输出，按 `s` 键回车后会使后面的滚动不再停顿，按 `q` 键回车后跳出当前输出操作。如果 `n` 是负数，输出会每页滚动一次（即，每比当前屏幕的行数少一行时滚动）。默认值，`settings.scroll=0`，设定连续滚动。

在 `plain` 模块中定义的过程

```
string getstring(string name="", string default="", string prompt="",
                bool store=true);
int getint(string name="", int default=0, string prompt="",
           bool store=true);
real getreal(string name="", real default=0, string prompt="",
             bool store=true);
pair getpair(string name="", pair default=0, string prompt="",
             bool store=true);
triple gettriple(string name="", triple default=(0,0,0), string prompt="",
                 bool store=true);
```

可以用来显示提示符并从 `stdin` 中读入值，它使用 GNU `readline` 库。如果 `store=true`，名字 `name` 的值的历史记录会被存在文件 `".asy_history_" + name` 中（见手册的历史记录一节<sup>48</sup>）。历史记录中最近的值将提供随后执行的默认值。默认值（初始为 `default`）在 `prompt` 之后显示。这些函数都基于内部过程

```
string readline(string prompt="", string name="", bool tabcompletion=false);
void saveline(string name, string value, bool store=true);
```

这里，`readline` 使用按照 `prompt` 格式化的初始值提示用户输入，而 `saveline` 用于把字符串 `value` 保存在一个名为 `name` 的本地历史记录，可选地把本地历史记录保存在一个文件 `".asy_history_" + name` 中。

过程 `history(string name, int n=1)` 可以用来查找最近为字符串 `name` 输入的第 `n` 个值（或直到 `historylines` 的所有值，如果 `n=0`）。过程 `history(int n=0)` 返回交互历史。例如，

```
write(output("transcript.asy"),history());
```

将交互历史输出到文件 `transcript.asy`。

函数 `int delete(string s)` 删去文件名为字符串的 `s` 文件。除非 `-globalwrite`（或 `-nosafe`）选项可用，否则此文件必须存在于当前目录。函数 `int rename(string from, string to)` 可以用于把文件 `from` 改名为文件 `to`。除非 `-globalwrite`（或 `-nosafe`）选项可用，否则此操作也限定在当前目录。函数

<sup>48</sup>这个节译本并不包括此节。请参看英文版手册 9 Interactive mode 一章的相关内容。——译者注

```
int convert(string args="", string file="", string format="");
int animate(string args="", string file="", string format="");
```

分别调用 ImageMagick 命令 `convert` 和 `animate`，附带参数 `args` 以及由字符串 `file` 和 `format` 构成的文件名<sup>49</sup>。如果设置 `safe` 为假，则函数 `int system(string s)` 可以用来执行任意的系统命令 `s`。

## 6.7 变量初始式

一个变量可以在声明时赋值，就像 `int x=3;`，其中变量 `x` 被赋以值 3。与字面常量如 3 一样，任意表达式都可以用于初始式，如 `real x=2*sin(pi/2);`。

一个变量在初始式被求值之前不会加入名字空间之中，因此举例而言，在

```
int x=2;
int x=5*x;
```

中，第二行初始式中的 `x` 表示的是在第一行声明的变量 `x`。于是第二行声明了一个变量 `x` 遮蔽了原来的 `x` 并将其初始化为值 10。

大多数类型的变量声明时可以没有明确的初始式，并且它们会被该类型的默认初始式初始化：

数值类型 `int`、`real` 和 `pair` 的变量都初始化为零；类型 `triple` 的变量初始化为 `0=(0,0,0)`。

`bool` 变量初始化为 `false`。

`string` 变量初始化为空串。

`transform` 变量初始化为恒同变换。

`path` 和 `guide` 变量初始化为 `nullpath`。

`pen` 变量初始化为默认画笔。

`frame` 和 `picture` 变量分别初始化为空的帧与图。

`file` 变量初始化为 `null`。

用户定义的数组、结构体和函数类型的默认初始式分别在对应的章节说明。一些类型，如 `code`，没有默认的初始式。当一个这种类型的变量被引入，用户必须明确用一个值对其进行初始化。

任意类型 `T` 的默认初始式可能通过定义函数 `T operator init()` 来重新声明。例如，`int` 变量通常被初始化为零，但在

```
int operator init() {
    return 3;
}

int y;
```

<sup>49</sup> 这可能是不安全的，因为不能保证这两个命令是不是来自 ImageMagick 以及它们会做什么。特别地，Windows 系统把 FAT 硬盘格式转换为 NTFS 硬盘格式的命令就是 `convert`。——译者注

中，变量 `y` 初始化为 3。这只是一个用于说明的例子；并不推荐重新声明内部类型的初始式。`operator init` 典型的用法是定义用户定义类型的有意义的默认值。

## 6.8 结构体

用户可以定义自己的数据类型如结构体，连同用户定义的算子，就像在 C++ 中一样。默认地，结构体的成员是 `public`（公有的；可以在代码的任何地方读和更改），但也可选择声明为 `restricted`（受限的；可以在任何地方读，但只能在定义它的结构体内部修改）或是 `private`（私有的；仅能在结构体内部读写）。在一个结构体的定义中，关键字 `this` 可用于在一个表达式中引用包围它的结构体。在结构体顶层作用域的任何代码都会在初始化时执行。

变量保存的是结构体的引用<sup>50</sup>。就是说，例如<sup>51</sup>：

```
struct T {
    int x;
}
T foo=new T;
T bar=foo;
bar.x=5;
```

变量 `foo` 保存一个到结构体 `T` 的实例的引用。当 `bar` 被赋值为 `foo` 时，它也保存一个和 `foo` 相同的引用。赋值 `bar.x=5` 改变那个实例中 `x` 域的值，因而 `foo.x` 将也等于 5。

表达式 `new T` 创建结构体 `T` 的一个新的实例并返回到该实例的引用。在建立新实例时，在记录定义的体中的所有代码都会被执行。例如

```
int Tcount=0;
struct T {
    int x;
    ++Tcount;
}
T foo=new T;
```

这里，表达式 `new T` 将会产生这个类的一个新的实例，但同时也会使 `Tcount` 增长，因此它会保有实例产生数目的轨迹<sup>52</sup>。

表达式 `null` 可以转换为任何结构体类型的空引用，即一个不指向任何实际结构体实例的引用。试图使用一个空引用的域将导致错误。

函数 `bool alias(T,T)` 检查两个结构体是否引用结构体的同一个实例（或同时为 `null`）。例如，在本节开头的示例代码中，`alias(foo,bar)` 将返回真，但 `alias(foo,new T)` 将返回假，因为 `new T`

<sup>50</sup> 请读者注意，这与 C/C++ 有重要的不同。——译者注

<sup>51</sup> 一个语法细节：结构体的定义后不需要再加分号，这与 C/C++ 不同。——译者注

<sup>52</sup> 然而这个用法并不很好。如果真要记录实例创建的数目，应该将 `Tcount` 设置为类 `T` 的静态成员。参看 84 页 6.15 节。——译者注

创建结构体 `T` 的一个新的实例。布尔运算符 `==` 和 `!=` 默认分别等价于 `alias` 和 `!alias`，但对特定的类型它们可以被改写（例如做深比较<sup>53</sup>）。

在定义结构体 `T` 之后，一个类型为 `T` 的变量默认会被初始化为一个新的实例（`new T`）。然而，在结构体的定义中，类型为 `T` 的变量默认会被初始化为 `null`。这个特殊的行为可以避免在下面这种代码中建立一个新实例时的无穷递归：

```
struct tree {
    int value;
    tree left;
    tree right;
}
```

这里是一个简单的例子，解释结构体的使用：

```
struct S {
    real a=1;
    real f(real a) {return a+this.a;}
}

S s;                      // 用 new S 初始化 s

write(s.f(2));             // 输出 3

S operator + (S s1, S s2)
{
    S result;
    result.a=s1.a+s2.a;
    return result;
}

write((s+s).f(0));         // 输出 2
```

有构造一个结构体的新实例的函数通常很方便。比如我们有一个 `Person` 结构体：

```
struct Person {
    string firstname;
    string lastname;
}

Person joe=new Person;
```

<sup>53</sup>指按结构体的嵌套结构递归进行的，逐个域的比较。相关的概念还有深复制。在后面章节数组的讨论中，也有类似的深比较和深复制概念。——译者注



```
joe.firstname="Joe";
joe.lastname="Jones";
```

创建一个新的“人”（Person）是件繁琐的事；要用三行来建立一个新的实例并初始化它的域（尽管这比在实际生活中建立一个新的人花的力气还要少得多）。

我们可以通过定义一个构造函数 `Person(string, string)` 来简化这件工作：

```
struct Person {
    string firstname;
    string lastname;

    static Person Person(string firstname, string lastname) {
        Person p=new Person;
        p.firstname=firstname;
        p.lastname=lastname;
        return p;
    }
}

Person joe=Person.Person("Joe", "Jones");
```

现在比以前建立一个新的实例要容易了，但我们还得用限定过的名字 `Person.Person` 引用构造函数。如果紧接着结构体的定义后增加一行

```
from Person unravel Person;
```

则构造函数就可以不加限定地使用：`Person joe=Person("Joe", "Jones");`。

现在构造函数是容易用了，可定义起来却成了件麻烦事儿。如果你要写大量的构造函数，你就会发现每次你都得重复大量的代码。幸运的是，你友好的邻居 Asymptote 的开发者们设计出了一种方法能自动化完成大部分的过程。

如果，在结构体的（定义）体中，Asymptote 遇到形如 `void operator init(args)` 的函数定义，则会隐式地定义一个以 `args` 为函数的构造函数来初始化一个结构体的新实例。即，它实质上定义了如下的构造函数（假定结构体名叫 `Foo`）：

```
static Foo Foo(args) {
    Foo instance=new Foo;
    instance.operator init(args);
    return instance;
}
```

这个构造函数也会被隐式地复制到结构体定义结束后的外围作用域中，因而它可以随后就不加限定地以结构体的名字使用。我们 `Person` 的例子从而可以实现为：



```

struct Person {
    string firstname;
    string lastname;

    void operator init(string firstname, string lastname) {
        this.firstname=firstname;
        this.lastname=lastname;
    }
}

Person joe=Person("Joe", "Jones");

```

`operator init` 隐式定义构造函数的用法不应与它定义变量的默认值的用法相混淆（见 57 页 6.7 节）。确实，在前一种情况，`operator init` 的返回值类型必须是 `void`，而第二种情形它必须是变量的（非 `void`）类型。

函数 `cputime()` 返回一个结构体 `cputime`，它保存不断增长的 CPU 时间，分为域 `parent.user`, `parent.system`, `child.user` 和 `child.system`。为了方便，增加的域 `change.user` 和 `change.system` 显示对应的总父进程和子进程 CPU 时间自上次调用 `cputime()` 的改变量。函数

```

void write(file file=stdout, string s="", cputime c,
           string format=cputimeformat, suffix suffix=none);

```

显示增加的用户 CPU 时间（后跟“u”）、增加的系统 CPU 时间（后跟“s”）、总的用户 CPU 时间（后跟“U”）以及总的系统 CPU 时间（后跟“S”）。

很像 C++ 中那样，类型转换（见 80 页 6.13 节）提供了结构体<sup>54</sup>继承和虚函数的一个优雅的实现<sup>55</sup>：

```

struct parent {
    real x;
    void operator init(int x) {this.x=x;}
    void virtual(int) {write(0);}
    void f() {virtual(1);}
}

void write(parent p) {write(p.x);}

struct child {

```

<sup>54</sup>即 C++ 中的类。——译者注

<sup>55</sup>这只是对于 C++ 继承和多态性的一种模拟，然而这或许并不能算是如何优雅的实现。对这里列出的简单示例，确实可以进行一些改进：在 `child` 定义中的 `parent` 声明后面加一句 `unravel parent`；可以免去在引用基类成员 `parent.x` 时的限定符 `parent`。而对于虚函数的定义，或许 `_f(parent obj=this, int)` 比 `virtual(int)` 的写法更好，因为虚函数可能不只一个，并且很可能会用到 `this` 变量引用。——译者注

```

parent parent;
real y=3;
void operator init(int x) {parent.operator init(x);}
void virtual(int x) {write(x);}
parent.virtual=virtual;
void f()=parent.f;
}

parent operator cast(child child) {return child.parent;}

parent p=parent(1);
child c=child(2);

write(c);                      // 输出 2;

p.f();                          // 输出 0;
c.f();                          // 输出 1;

write(c.parent.x);              // 输出 2;
write(c.y);                     // 输出 3;

```

更多结构体的例子，见 Asymptote 基础模块 plain 中的 `Legend` 和 `picture`。

## 6.9 运算符

### 6.9.1 算术和逻辑运算符

Asymptote 使用标准二元算术运算符。然而，当一个整数被另一个整数所除时，两个参数都会在相除之前被转换为实数值并返回一个实数商（鉴于这通常是人们想要的）。函数 `int quotient(int x, int y)` 返回小于等于  $x/y$  的最大整数。在所有其他情况下，两个运算对象会被提升为相同的类型，此类型也将成为结果的类型：

+	加法
-	减法
*	乘法
/	除法
%	模（求余数）；结果总与除数的符号相同。特别地，对所有整数 $p$ 和非零整数 $q$ 得到结果 <code>q*quotient(p,q)+p%q == p</code> 。
^	乘方；如果指数（第二个参数）是一个整数型，则会使用递归乘法；否则，会使用对数和指数计算（** 与 ^ 同义）。

通常的布尔运算符同样有定义<sup>56</sup>：

<code>==</code>	等于
<code>!=</code>	不等于
<code>&lt;</code>	小于
<code>&lt;=</code>	小于等于
<code>&gt;=</code>	大于等于
<code>&gt;</code>	大于
<code>&amp;&amp;</code>	与（右边参数为条件求值 <sup>57</sup> ）
<code>&amp;</code>	与
<code>  </code>	或（右边参数为条件求值）
<code> </code>	或
<code>^</code>	异或
<code>!</code>	非

Asymptote 同样支持类似 C 的条件语法：

```
bool positive=(pi >= 0) ? true : false;
```

函数 `T interp(T a, T b, real t)` 对所有非整数的内部数值类型 `T` 返回  $(1-t)*a+t*b$ 。如果 `a` 和 `b` 是画笔（pen），它们会首先被提升到相同的颜色空间。

Asymptote 同样定义了按位函数 `int AND(int,int)`, `int OR(int,int)`, `int XOR(int,int)` 以及 `int NOT(int)`。

## 6.9.2 自（赋值）与前缀运算符

与在 C 中一样，每个运算符 `+`, `-`, `*`, `/`, `%` 和 `^` 都可以作为自（赋值）运算符使用。前缀运算符 `++`（加一）与 `--`（减一）也有定义。例如，

```
int i=1;
i += 2;
int j=++i;
```

就等价于代码

```
int i=1;
i=i+2;
int j=i+1;
```

然而，后缀运算符如 `i++` 和 `i--` 并未定义（由于 `--` 路径连接运算符产生的固有歧义性）。只在罕见的实例中 `i++` 和 `i--` 才真正需要，它们可以分别替换为表达式 `(++i-1)` 和 `(--i+1)`。

<sup>56</sup>这里的布尔运算符都只能用于 `bool` 类型量，不包括按位计算。而且与 C/C++ 不同，其他类型的量不能依是否为 0 值自动转化为 `bool` 类型的量。因而 `3 && 5` 这种写法是非法的，必须写成 `3!=0 && 5!=0`。——译者注

<sup>57</sup>指短路求值。仅当左边参数求值为真时才对右边求值。或运算类似，仅当左边参数求值为假时才对右边求值。——译者注

### 6.9.3 用户自定义运算符

下列的符号可以与 `operator` 一起为结构体或内部类型定义或重定义运算符：

```
- + * / % ^ ! < > == != <= >= & | ^~ .. :: -- --- ++
<< >> $ $$ @ @@
```

第二行的运算符拥有比布尔运算符 `<`, `>`, `<=` 和 `>=` 高一级的优先级<sup>58</sup>。

路向运算符如 `..` 可以重载，比如说，写一个用户自定义函数从给定路向产生一条新的路向：

```
guide dots(... guide[] g)=operator ..;

guide operator ..(... guide[] g) {
    guide G;
    if(g.length > 0) {
        write(g[0]);
        G=g[0];
    }
    for(int i=1; i < g.length; ++i) {
        write(g[i]);
        write();
        G=dots(G,g[i]);
    }
    return G;
}

guide g=(0,0){up}..{SW}(100,100){NE}..{curl 3}(50,50)..(10,10);
write("g=",g);
```

## 6.10 隐式放缩

如果一个数值字面量在特定类型的表达式前面，则它们会相乘：

```
int x=2;
real y=2.0;
real cm=72/2.540005;

write(3x);
```

<sup>58</sup>手册这一节对于运算符及其自定义的介绍还是有些过于简略了，比如所有运算符的优先级和结合性都没有详细说明，事实上它们大致与 C/C++ 中相同。第二行的运算符对内部类型都没有定义，据实验，它们都是左结合的普通二元运算符。还有需要说明的是，前置的 `++`, `--` 运算符的运算对象必须是一个左值，这会在重载时带来一些限制。C/C++ 中常见的逗号运算符、关于指针的运算符、`sizeof` 等都不存在。赋值、条件、圆点、指标 `[]`、括号等也都不能重载。重载 `**` 与重载 `^` 效果相同。——译者注

```

write(2.5x);
write(3y);
write(-1.602e-19 y);
write(0.5(x,y));
write(2x^2);
write(3x+2y);
write(3(x+2y));
write(3sin(x));
write(3(sin(x))^2);
write(10cm);

```

这会产生输出

```

6
5
6
-3.204e-19
(1,1)
8
10
18
2.72789228047704
2.48046543129542
283.464008929116

```

## 6.11 函数

Asymptote 的函数被看作是带有标识的变量（非函数变量有空的标识）。允许有名字相同的变量，只要它们有不同的标识。

函数的参数是传值的。要以传引用方式传递参数，只不过需要把它用一个结构体包装起来（见 58 页 6.8 节）。

这里是 Asymptote 函数的一些重要特性：

1. 带标识的变量（函数）和不带标识的变量（非函数变量）是不同的：

```

int x, x();
x=5;
x=new int() {return 17;};
x=x();          // 调用 x() 并将结果 17 存入标量 x

```

## 2. 允许使用传统函数的定义：

```
int sqr(int x)
{
    return x*x;
}
sqr=null;    // 然而这个函数仍然只是一个变量。
```

## 3. 类型转换可用于消解歧义：

```
int a, a(), b, b();    // 合法：创建四个变量
a=b;                  // 不合法：赋值有歧义
a=(int) b;            // 合法：消解歧义
(int) (a=b);          // 合法：消解歧义
(int) a=b;            // 不合法：类型转换表达式不能为左值

int c();
c=a;                  // 合法：只有一种可能的赋值
```

## 4. 同样允许匿名（又称“高阶”）函数：

```
typedef int intop(int);
intop adder(int m)
{
    return new int(int n) {return m+n;};
}
intop addby7=adder(7);
write(addby7(1));    // 写入 8
```

5. 可以通过赋值为另一个（匿名的或命名的）函数来重定义一个函数 **f**，即便对那些以前声明过的函数的调用。然而，如果 **f** 被新的函数定义重载，先前的调用仍会访问 **f** 原来的版本，如下例所示<sup>59</sup>：

```
void f() {
    write("hi");
}

void g() {
    f();
}
```

<sup>59</sup>这两句话有些拗口。注意这里有一个区别，即原来只是声明而没有定义（而且没有通过自定义 `operator init()` 而初始化，从而初值为 `null`）的函数 **f**，如果被其他函数 **g** 的定义调用，则给 **f** 合适的定义后，**g** 的定义也会跟着变化；但如果原来 **f** 已经有定义，则重定义 **f** 并不会改变 **g** 的定义，正如例子中展示的那样。这儿的例子只展示了后一种情况，前一种情况例子可以参看下一个条目。——译者注

```

}

g(); // 写入 "hi"

f=new void() {write("bye");};

g(); // 写入 "bye"

void f() {write("overloaded");};

f(); // 写入 "overloaded"
g(); // 写入 "bye"

```

6. 匿名函数可以用于重定义一个已经声明（并隐式初始化为空函数（`null`））但尚未明确定义的函数变量：

```

void f(bool b);

void g(bool b) {
    if(b) f(b);
    else write(b);
}

f=new void(bool b) {
    write(b);
    g(false);
};

g(true); // 写入 true, 然后写入 false.

```

Asymptote 是我们已知唯一的把函数看做变量，但又通过基于标识的变量而允许重载的语言。

允许函数递归调用自身。像在 C++ 中一样，无穷嵌套的递归将导致栈溢出（报告为一个段错误，除非一个完整工作着的 GNU libsigsegv 库（如 2.4 或更新）在配置的时候已经安装）。

### 6.11.1 默认参数

Asymptote 支持比 C++ 更为灵活的默认函数参数机制：它们可以在函数原型的任何位置出现。由于特定的数据类型被隐式转换为更复杂的类型（见 80 页 6.13 节），经常可以通过将函数参数从最简单到最复杂排列来避免歧义性。例如，给定

```
real f(int a=1, real b=0) {return a+b;}
```

则 `f(1)` 返回 1.0, 但 `f(1.0)` 返回 2.0。

默认参数的值在被调函数定义的环境中对给定 `Asymptote` 表达式求值决定。

### 6.11.2 具名参数

有时很难记住函数声明中参数出现的顺序。具名（带关键字的）参数使调用多参数函数更容易了。不像在 C 和 C++ 语言里那样，在函数参数中的赋值被解释为在函数标识中同名参数的赋值，而不在局部作用域中。命令行选项 `-d` 可用于检查 `Asymptote` 代码中可能把具名参数误拼为一个局部赋值的情况。

当参数与标识匹配时，首先所有的关键字相匹配，而后不具名的参数再如常与未匹配的形参相匹配。例如，

```
int f(int x, int y) {
    return 10x+y;
}
write(f(4,x=3));
```

输出 34，因为在我们试图给无名参数 4 匹配时，`x` 已经匹配过了，因而它与下一项 `y` 来匹配。

对于想要在函数参数中进行局部变量赋值的罕见情形（通常不是一种好的编程实践），仅仅需要把赋值放进括号里。例如，给定在前面例子中 `f` 的定义，

```
int x;
write(f(4,(x=3)));
```

就等价于语句

```
int x;
x=3;
write(f(4,3));
```

而输出 43。

作为一个技术细节，我们指出，鉴于在同一个作用域中允许有同名但有不同的标识的变量，代码

```
int f(int x, int x()) {
    return x+x();
}
int seven() {return 7;}
```

在 `Asymptote` 中是合法的，而 `f(2,seven)` 返回 9。一个具名参数匹配第一个同名的未匹配的形参，于是 `f(x=2,x=seven)` 就是等价的调用，然而 `f(x=seven,2)` 则不是，因为第一个参数与第一个形参匹配，而 `int()` 不能隐式地转换为 `int`。默认参数不会影响一个具名参数匹配的形式参数，于是如果 `f` 被定义为



```
int f(int x=3, int x()) {
    return x+x();
}
```

则 `f(x=seven)` 将是非法的，即使 `f(seven)` 显然是允许的。

### 6.11.3 剩余参数

剩余参数允许人们写使用可变个数参数的函数：

```
// 此函数将其参数求和
int sum(... int[] nums) {
    int total=0;
    for(int i=0; i < nums.length; ++i)
        total += nums[i];
    return total;
}

sum(1,2,3,4);           // 返回 10
sum();                   // 返回 0

// 此函数用第一个参数减去后面的参数
int subtract(int start ... int[] subs) {
    for(int i=0; i < subs.length; ++i)
        start -= subs[i];
    return start;
}

subtract(10,1,2);        // 返回 7
subtract(10);             // 返回 10
subtract();               // 非法
```

把一个参数放进一个剩余表称作打包。可以为剩余参数给定一个明确的参数表，从而 `subtract` 也可以选择实现为

```
int subtract(int start ... int[] subs) {
    return start - sum(... subs);
}
```

甚至可以把剩余参数与普通参数组合使用：

```
sum(1,2,3 ... new int[] {4,5,6}); // 返回 21
```

它建立一个六个元素的数组，以 `nums` 传递给 `sum`。相反的操作，解包，却是不允许的：

```
subtract(... new int[] {10, 1, 2});
```

是非法的，因为开头的形参不匹配。

如果没有参数被打包，一个长度为零的数组（与 `null` 相反<sup>60</sup>）会约束到剩余参数上。注意剩余参数的默认参数会被忽略而且剩余参数不会约束到一个关键字上。

在 `Asymptote` 中重载的解决与 C++ 中使用的函数匹配规则类似。每个参数匹配会被给以一个得分。精确匹配的得分比需要类型转换的得分好，而形参的匹配（不考虑类型转换）又比把一个参数打包进剩余数组的得分好。如果有唯一的最大候选，它就会被选取；否则，就会有歧义性错误。

```
int f(path g);
int f(guide g);
f((0,0)--(100,100)); // 匹配第二个；参数是一个 guide

int g(int x, real y);
int g(real x, int x);

g(3,4); // 歧义；第一个候选的第一个参数更好，
        // 但第二个候选的第二个参数更好

int h(... int[] rest);
int h(real x ... int[] rest);

h(1,2); // 第二个定义匹配，尽管有一个类型转换，
        // 因为类型转换比打包更好

int i(int x ... int[] rest);
int i(real x, real y ... int[] rest);

i(3,4); // 歧义；第一个候选的第一个参数更好，
        // 但第二个候选的第二个参数更好
```

#### 6.11.4 数学函数

`Asymptote` 有标准 `libm` 数学 `real(real)` 型函数的内部版本：`sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `exp`, `log`, `pow10`, `log10`, `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`, `sqrt`, `cbrt`, `fabs`, `expm1`, `log1p` 以及

<sup>60</sup>长度为零的数组（空白数组 `empty array`）和空数组（`null array`）是不同的概念，见 72 页 6.12 节的解释。事实上，长度为零的数组是已分配空间的数组，而空数组 `null` 未分配空间，意义相当于 C/C++ 中的空指针。——译者注

恒等函数 `identity`<sup>61</sup>。Asymptote 同样定义了  $n$  阶 Bessel 函数：第一类 `Jn(int n, real)` 以及第二类 `Yn(int n, real)`，还有  $\Gamma$  函数 `gamma`，误差函数 `erf`，以及互补误差函数 `erfc`。还包括标准的 `real(real, real)` 型函数 `atan2`, `hypot`, `fmod`, `remainder`<sup>62</sup>。

函数 `degrees(real radians)` 和 `radians(real degrees)` 可用来在弧度和度之间转换。函数 `Degrees(real radians)` 返回在区间  $[0, 360)$  中的角度度数。为方便计，Asymptote 定义了使用度而非弧度的标准三角函数变体 `Sin`, `Cos`, `Tan`, `aSin`, `aCos` 和 `aTan`。我们同样定义了 `sqrt`, `sin`, `cos`, `exp`, `log` 和 `gamma` 的复数版本。

函数 `floor`, `ceil` 和 `round`<sup>63</sup>与它们通常的定义不同，它返回一个整数值而非实数值（鉴于这是通常人们想要的）。函数 `Floor`, `Ceil` 和 `Round` 分别与前面的类似，只是如果结果不能被转化为一个合法的整数值，它们会对正参数返回 `intMax` 而对负参数返回 `intMin`，而不会产生整数溢出。我们同样定义了函数 `sgn`，它以一个整数（-1, 0 或 1）返回实数参数的符号。

有 `abs(int)` 函数，还有 `abs(real)` 函数（等价于 `fabs(real)`），以及 `abs(pair)` 函数（等价于 `length(pair)`）。

随机数可由 `srand(int)` 设定种子并由 `int rand()` 产生，它返回一个 0 到整数 `randMax` 的整数。函数 `unitrand()` 返回一个  $[0, 1]$  区间均匀分布的随机数。Gauss 分布的随机数生成器 `Gaussrand` 以及一集统计过程，包括 `histogram`（直方图），由基础模块文件 `stats.asy` 提供<sup>64</sup>。还定义了函数<sup>65</sup>`factorial(int n)`，返回  $n!$ ，以及 `choose(int n, int k)`，返回  $n!/(k!(n-k)!)$ 。

如果配置加上了 GNU 科学计算库（GNU Scientific Library, GSL），它在 <http://www.gnu.org/software/gsl/> 可得到，则 Asymptote 会包括一个内部模块 `gsl` 定义了 Airy 函数 `Ai(real)`, `Bi(real)`, `Ai_deriv(real)`, `Bi_deriv(real)`, `zero_Ai(int)`, `zero_Bi(int)`, `zero_Ai_deriv(int)`, `zero_Bi_deriv(int)`，Bessel 函数 `I(int, real)`, `K(int, real)`, `j(int, real)`, `y(int, real)`, `i_scaled(int, real)`, `k_scaled(int, real)`, `J(real, real)`, `Y(real, real)`, `I(real, real)`, `K(real, real)`, `zero_J(real, int)`，椭圆函数 `F(real, real)`, `E(real, real)` 和 `P(real, real)`，指数/三角积分 `Ei`, `Si` 和 `Ci`，Legendre 多项式 `Pl(int, real)`，以及 Riemann  $\zeta$  函数 `zeta(real)`。例如，要计算 1.0 的正弦积分 `Si`：

```
import gsl;
write(Si(1.0));
```

Asymptote 同样提供了一些一般目的的数值过程：

```
real newton(int iterations=100, real f(real), real fprime(real), real x,
            bool verbose=false);
```

<sup>61</sup> 三角函数 `sin` 正弦，`cos` 余弦，`tan` 正切以弧度为单位；反三角函数 `asin` 反正弦，`acos` 反余弦，`atan` 反正切返回弧度值；指数 `exp`，对数 `log` 以  $e$  为底；指数 `pow10`，对数 `log10` 以 10 为底；双曲（反）三角函数 `sinh` 双曲正弦，`cosh` 双曲余弦，`tanh` 双曲正切，`asinh` 双曲反正弦，`acosh` 双曲反余弦，`atanh` 双曲反正切；`sqrt` 平方根；`cbirt` 立方根；`fabs` 实数绝对值；`expm1` 为  $x \mapsto \exp x - 1$ （对接近零的参数也精确）；`log1p` 为  $x \mapsto \log(1 + x)$ （对接近零的参数也精确）；恒等函数 `identity` 与产生单位方阵的函数同名。——译者注

<sup>62</sup> 函数 `atan2(real y, real x)` 计算  $y/x$  的反正切；函数 `hypot(real x, real y)` 计算三角形斜边长  $\sqrt{x^2 + y^2}$ ；`fmod` 和 `remainder` 都是实数的求余函数，前者符号与被除数相同，后者取对称模。——译者注

<sup>63</sup> `floor` 是向下取整  $\lfloor x \rfloor$ ，`ceil` 是向上取整  $\lceil x \rceil$ ，而 `round` 是四舍五入取整。——译者注

<sup>64</sup> 这里的介绍十分简略，有需要的用户应去参考 `stats.asy` 的源文件以获得详细的函数参数形式。——译者注

<sup>65</sup> 这两个函数都返回 `int` 类型。——译者注

给定导数 `fprime` 和初值 `x`，使用 Newton-Raphson 迭代求解实值可微函数 `f` 的根。如果 `verbose=true`，每次迭代的诊断值将打印出来。如果在最大允许循环次数（`iterations`）之后迭代失败，将返回 `realMax`。

```
real newton(int iterations=100, real f(real), real fprime(real), real x1,
           real x2, bool verbose=false);
```

给定导数 `fprime`，使用包围的 Newton-Raphson 二分法（bracketed Newton-Raphson bisection）在区间 `[x1, x2]`（在上面 `f` 的端点值有相反的符号）求解实值可微函数 `f` 的根。如果 `verbose=true`，每次迭代的诊断值将打印出来。如果在最大允许循环次数（`iterations`）之后迭代失败，将返回 `realMax`。

```
real simpson(real f(real), real a, real b, real acc=realEpsilon, real dxmax=b-a)
    返回使用 Simpson 自适应法计算的 f 从 a 到 b 的积分。
```

## 6.12 数组

在内部或用户自定义的类型后加上 `[]` 就得到数组。数组 `A` 的元素 `i` 可用 `A[i]` 访问。默认情况下，试图使用负指标对一个数组元素进行访问或赋值将产生错误。以超出数组长度的指标读数组元素同样会产生错误；然而，对超出数组长度处的元素赋值会导致数组变长以容纳新元素。可以用一个整数数组 `B` 来作为数组 `A` 的指标：数组 `A[B]` 由数组 `A` 以 `B` 中各元素为指标构成。对整个数组元素迭代，有一种方便的 Java 风格的简化形式；见第 19 页数组迭代部分<sup>66</sup>。

声明

```
real[] A;
```

初始化 `A` 为一个空白（长度为零的）数组（empty array）。空白数组应与空数组（null array，或译“无效数组”更直白）区别。如果我们说

```
real[] A=null;
```

则 `A` 完全不能被解引用（空数组根本没有长度并且不能从中读取或是向其元素赋值）。

数组可以像这样明确地初始化：

```
real[] A={0,1,2};
```

在 `Asymptote` 中数组的赋值进行的是浅复制：只有指针被复制过来（如果一个副本被修改，则另一个也同样会被修改）。后面列出的 `copy` 函数提供了数组的深复制。

每个类型为 `T[]` 的数组 `A` 都有虚成员

```
int length,
```

```
void cyclic(bool b),
```

<sup>66</sup>指 `for (elem : array) { statements }` 的格式。在本章开头。——译者注

```

bool cyclicflag,

int[] keys,

T push(T x),

void append(T[] a),

T pop(),

void insert(int i ... T[] x),

void delete(int i, int j=i),

void delete(), 以及

bool initialized(int n)。

```

成员 `A.length` 计算数组的长度。设置 `A.cyclic(true)` 表示指标应该约化为对当前数组长度取模的结果。从一个非空的循环 (cyclic) 数组中读写不会造成指标越界错误或是数组长度变化。成员 `A.cyclicflag` 返回对于 `cyclic` 标记的当前设置。

成员 `A.keys` 求出一个整数数组，它包含原数组中已初始化的项的指标，按升序排列。所以，对一个长度为 `n` 并且所有项均初始化的数组，`A.keys` 将求得从 0 到 `n-1` 的闭区间整数数组。每当 `A.keys` 求值时，都会求出一个新的键值 (key) 数组。

函数 `A.push` 和 `A.append` 把参数添加到数组的末尾，而 `A.insert(int i ... T[] x)` 把 `x` 插入数组指标 `i` 的位置。为方便 `A.push` 返回压入的项目。函数 `A.pop()` 弹出并返回最后一个元素，而 `A.delete(int i, int j=i)` 删除指标在 `[i, j]` 范围内的元素，并把指标较大的元素都移回前面。如果不给参数，则 `A.delete()` 提供了一种方便的方式删去 `A` 的所有元素。过程 `A.initialized(int n)` 可用于检查指标 `n` 处元素是否已初始化。如同所有 `Asymptote` 函数一样，`cyclic`, `push`, `append`, `pop`, `insert`, `delete` 和 `initialized` 可以从数组“脱下”并作用于它们自身。例如，

```

int[] A={1};
A.push(2);           // A 现在包含 {1,2}
A.append(A);         // A 现在包含 {1,2,1,2}
int f(int)=A.push;
f(3);               // A 现在包含 {1,2,1,2,3}
int g()=A.pop;
write(g());         // 输出 3
A.delete(0);        // A 现在包含 {2,1,2}
A.delete(0,1);      // A 现在包含 {2}
A.insert(1,3);      // A 现在包含 {2,3}
A.insert(1 ... A);  // A 现在包含 {2,2,3,3}
A.insert(2,4,5);    // A 现在包含 {2,2,4,5,3,3}

```

[] 后缀同样可以出现在变量名的后面；有时这对声明同一类型的一系列变量和数组比较方便：

```
real a,A[];
```

这声明了 `a` 为 `real` 并隐式地声明 `A` 为类型 `real[]`。

在下列的内部数组函数中，`T` 表示一个一般的类型。注意内部函数 `alias`, `array`, `copy`, `concat`, `sequence`, `map` 和 `transpose`，依赖于函数 `T`，仅在初次声明一个类型为 `T[]` 的变量后才有定义。

```
new T[]
```

返回类型为 `T[]` 的新的空白数组。

```
new T[] {list}
```

返回一个类型为 `T[]` 的新数组，由 `list`（一个逗号分隔的元素列表）初始化。

```
new T[n]
```

返回一个新的 `n` 元数组。这 `n` 个数组元素不会被初始化，除非它们本身就是数组（在这种情况下它们每个都被初始化为空白数组）。

```
T[] array(int n, T value, int depth=intMax)
```

返回一个由 `value` 的 `n` 份复本组成的数组。默认情况下，如果 `value` 本身是一个数组，将对新数组的每个项做深复制。如果指明了 `depth`，这个深复制仅会递归指定的层数。

```
int[] sequence(int n)
```

如果 `n >= 1` 则返回数组 `{0,1,...,n-1}`（否则返回一个空数组）。

```
int[] sequence(int n, int m)
```

如果 `m >= n` 则返回数组 `{n,n+1,...,m}`（否则返回一个空数组）。

```
T[] sequence(T f(int), int n)
```

给定函数 `T f(int)` 和整数 `int n`，如果 `n >= 1` 则返回序列 `{f(i) : i=0,1,...,n-1}`（否则返回一个空数组）。

```
T[] map(T f(T), T[] a)
```

返回数组，它包含把函数 `f` 应用于数组 `a` 的每个元素的结果。这等价于 `sequence(new T(int i) {return f(a[i]);}, a.length)`。

```
int[] reverse(int n)
```

如果 `n >= 1` 则返回数组 `{n-1,n-2,...,0}`（否则返回一个空数组）。

```
int[] complement(int[] a, int n)
```

返回整数数组 `a` 对 `{0,1,2,...,n-1}` 的补集，于是 `b[complement(a,b.length)]` 就得到补集 `b[a]`。

```
real[] uniform(real a, real b, int n)
```

如果 `n >= 1` 则返回 `[a, b]` 的一个均匀划分，分为 `n` 段（否则返回一个空数组）<sup>67</sup>。

<sup>67</sup> 这个函数返回的是一个长为 `n+1` 的分点数组。——译者注

```
int find(bool[], int n=1)
```

返回第 `n` 个 `true` 值的指标，若找不到则返回 `-1`。如果 `n` 是负数，则从数组末尾开始倒着找第 `-n` 个值。

```
int search(T[] a, T key)
```

对于内部有序类型 `T`，在 `n` 元已排序的数组 `a` 中搜索一个区间包含 `key`，如果 `key` 比第一个元素还小则返回 `-1`，如果大于等于最后一个元素则返回 `n-1`，否则返回的指标对应区间的左（较小）端点。

```
T[] copy(T[] a)
```

返回数组 `a` 的一个深复制。

```
T[][] copy(T[][] a)
```

返回数组 `a` 的一个深复制。

```
T[][][] copy(T[][][] a)
```

返回数组 `a` 的一个深复制。

```
T[] concat(... T[][] a)
```

返回一个由给定参数拼接构成的新数组。

```
bool alias(T[] a, T[] b)
```

如果数组 `a` 和 `b` 是相同的<sup>68</sup>则返回 `true`。

```
T[] sort(T[] a)
```

对内部有序类型 `T`，返回 `a` 的增序排列的复本<sup>69</sup>。

```
T[][] sort(T[][] a)
```

对内部有序类型 `T`，返回 `a` 的复本，其各行按第一列排序，若相等则按下一列排序。例如：

```
string[][] a={{ "bob", "9"}, {"alice", "5"}, {"pete", "7"}, {"alice", "4"};
// 行排序（按第 0 列，使用第 1 列打破平衡）：
write(sort(a));
```

产生

```
alice    4
alice    5
bob      9
pete     7
```

```
T[] sort(T[] a, bool compare(T i, T j))
```

返回 `a` 的稳定递增排序复本，次序按如果 `compare(i, j)` 为真则元素 `i` 先于元素 `j` 定义。

<sup>68</sup>这里两个数组相同指它们的指针相同。——译者注

<sup>69</sup>也就是说，`a` 本身并不被修改。——译者注

`T[][] transpose(T[][] a)`

返回 `a` 的转置。

`T[][][] transpose(T[][][] a, int[] perm)`

返回 `a` 的 3 维转置，它由 `new int[]{0,1,2}` 的排列 `perm` 作用于每项的指标决定<sup>70</sup>。

`T sum(T[] a)`

对算术类型 `T`，返回 `a` 的和。在 `T` 是 `bool` 时，将返回 `a` 中为真的元素个数。

`T min(T[] a)`

`T min(T[][] a)`

`T min(T[][][] a)`

对内部有序类型 `T`，返回 `a` 中最小元素。

`T max(T[] a)`

`T max(T[][] a)`

`T max(T[][][] a)`

对内部有序类型 `T`，返回 `a` 中最大元素。

`T[] min(T[] a, T[] b)`

对内部有序类型 `T`，并且 `a` 和 `b` 长度相同，返回一个由 `a` 与 `b` 中对应元素较小值构成的数组。

`T[] max(T[] a, T[] b)`

对内部有序类型 `T`，并且 `a` 和 `b` 长度相同，返回一个由 `a` 与 `b` 中对应元素较大值构成的数组。

`pair[] pairs(real[] x, real[] y)`

对长度相同的数组 `x` 和 `y`，返回复数（序对 `pair`）数组

`sequence(new pair(int i) {return (x[i],y[i]);}, x.length)`。

`pair[] fft(pair[] a, int sign=1)`

返回 `a` 的快速 Fourier 变换（如果可选的 FFTW 包安装了的话），使用给定的符号 `sign`<sup>71</sup>。这是一个简单的例子：

```
int n=4;
pair[] f=sequence(n);
write(f);
pair[] g=fft(f,-1);
write();
```

<sup>70</sup>在数学上，这就是说把对称群  $S_3$  的一个元素（置换） $\sigma$  作用于  $\{a_{i_1, i_2, i_3} \mid 0 \leq i_1, i_2, i_3 < n\}$  每个元素下标的下标得到的结果  $\{a_{i_{\sigma(1)}, i_{\sigma(2)}, i_{\sigma(3)}} \mid 0 \leq i_1, i_2, i_3 < n, \sigma \in S_3\}$ 。——译者注

<sup>71</sup>当 `sign` 为 +1 时给出离散 Fourier 变换，当 `sign` 为 -1 时给出离散 Fourier 逆变换。——译者注



```
write(g);
f=fft(g,1);
write();
write(f/n);
```

`real dot(real[] a, real[] b)`  
返回向量  $a$  和  $b$  的点积。

`real[] tridiagonal(real[] a, real[] b, real[] c, real[] f)`  
求解周期三对角问题  $Lx = f$  并返回解  $x$ ，其中  $f$  为  $n$  维向量，而  $L$  为  $n \times n$  矩阵

$$\begin{pmatrix} b_0 & c_0 & & & a_0 \\ a_1 & b_1 & c_1 & & \\ & a_2 & b_2 & c_2 & \\ & & \ddots & \ddots & \ddots \\ c_{n-1} & & & a_{n-1} & b_{n-1} \end{pmatrix}$$

对 Dirichlet 边界条件（在此记为  $u_{-1}$  和  $u_n$ ），用  $f_0 - a_0 u_{-1}$  和  $f_{n-1} - c_{n-1} u_n$  替换  $f_0$ ，然后设  $a_0 = c_{n-1} = 0$ 。<sup>72</sup>

`real[] solve(real[][] a, real[] b, bool warn=true)`  
利用 LU 分解求解线性方程组  $ax = b$ ，并返回解  $x$ ，其中  $a$  是一个  $n \times n$  矩阵而  $b$  为长度为  $n$  的数组。例如：

```
import math;
real[][] a={{1,-2,3,0},{4,-5,6,2},{-7,-8,10,5},{1,50,1,-2}};
real[] b={7,19,33,3};
real[] x=solve(a,b);
write(a); write();
write(b); write();
write(x); write();
write(a*x);
```

如果  $a$  是奇异矩阵而 `warn` 为 `false`，返回一个空白矩阵。如果矩阵  $a$  是对角阵，过程 `tridiagonal` 提供了一种更为有效的算法（见前面 `tridiagonal` 的解释）。

`real[][] solve(real[][] a, real[][] b, bool warn=true)`  
求解线性方程组  $ax = b$  并返回解  $x$ ，其中  $a$  是一个  $n \times n$  矩阵而  $b$  是一个  $n \times m$  矩阵。如果  $a$  是奇异矩阵而 `warn` 为 `false`，返回一个空白矩阵。

`real[][] identity(int n)`  
返回  $n \times n$  单位矩阵。

<sup>72</sup>公式中的  $a, b, c, f$  分别对应数组 `a, b, c, f`。这里为方便公式书写改为数学记法。——译者注

```
real[] [] diagonal(... real[] a)
```

返回以 `a` 为对角线元素的对角矩阵。

```
real[] [] inverse(real[] [] a)
```

返回方阵 `a` 的逆。

```
real[] quadraticroots(real a, real b, real c)
```

这是一个数值鲁棒的求解器，返回二次方程  $ax^2 + bx + c = 0$  的实根，以升序排列。重根会分别列出。

```
pair[] quadraticroots(explicit pair a, explicit pair b, explicit pair c)
```

这是一个数值鲁棒的求解器，返回二次方程  $ax^2 + bx + c = 0$  的两个复根。

```
real[] cubicroots(real a, real b, real c, real d)
```

这是一个数值鲁棒的求解器，返回三次方程  $ax^3 + bx^2 + cx + d = 0$  的实根。重根会分别列出。

`Asymptote` 包含完整的一集向量化的数组算术（包括自赋值）命令与逻辑操作。为了计算速度需要，这些逐元素的命令由 C++ 代码实现。给定

```
real[] a={1,2};
real[] b={3,2};
```

则 `a == b` 与 `a >= 2` 都求值得到向量 `{false, true}`。要测试 `a` 和 `b` 的所有部分是否全都一致，可使用布尔函数 `all(a == b)`。同样可以使用条件表达式如 `(a >= 2) ? a : b`，它返回数组 `{3,2}`，或写 `(a >= 2) ? a : null`，它返回数组 `{2}`。

所有标志为 `real(real)` 的标准内部 `libm` 函数同样可以使用实数数组作为参数，效果如果隐式地调用了 `map`。

与其他内部类型一样，基本数据类型的数组可由赋值读入。在这个例子中，代码

```
file fin=input("test.txt");
real[] A=fin;
```

向 `A` 内读入实数值，直至达到文件末尾（或发生 I/O 错误）。如果用 `line(file)` 设置了行模式，则读入将改在达到行末时停止（行模式可以用 `line(file,false)` 清除）：

```
file fin=word(line(input("test.txt")));
real[] A=fin;
```

另一个有用的模式是逗号分隔值模式，由 `csv(file)` 设置而由 `csv(file,false)` 清除，它使得读入以逗号为界定符：

```
file fin=csv(input("test.txt"));
real[] A=fin;
```

要限定读入值的个数，使用 `dimension(file,int)` 函数：

```
file fin=input("test.txt");
real[] A=dimension(fin,10);
```

这会向 A 中读入 10 个值，除非先遇到文件末尾（或在行模式中遇到行末尾）。试图超出文件末尾读入将产生一个运行时错误信息。指定值为 0 的整数界限等价于在前面的例子中读入直至遇到文件末尾（或在行模式中遇到行末尾）。

基本数据类型的二维和三维的数组可以像这样读入：

```
file fin=input("test.txt");
real[] [] A=dimension(fin,2,3);
real[] [] [] B=dimension(fin,2,3,4);
```

同样地，一个零值的整数界限意味着没有限制。

有时数组维数与数据一起作为整数域保存在数组开头。这种数组可以用函数 `read1`, `read2` 和 `read3` 分别读入：

```
file fin=input("test.txt");
real[] A=read1(fin);
real[] [] B=read2(fin);
real[] [] [] C=read3(fin);
```

基本数据类型的一、二、三维数组可以用函数 `write(file,T[])`, `write(file,T[] [])`, `write(file,T[] [] [])` 分别输出。

### 6.12.1 切割

Asymptote 允许使用一种类似 Python 的语法以切割（slice）来表示数组的一个片断。如果 A 是一个数组，表达式 `A[m:n]` 返回一个新数组，它由 A 中指标从 m 开始直到但不包括 n 的元素构成。例如，

```
int[] x={0,1,2,3,4,5,6,7,8,9};
int[] y=x[2:6];           // y={2,3,4,5};
int[] z=x[5:10];         // z={5,6,7,8,9};
```

如果省略左边的指标，则取为 0。如果省略右边的指标则取为数组长度。如果二者都省略了，则切割从数组开头直到末尾，产生原数组的一个非循环的深复本。例如：

```
int[] x={0,1,2,3,4,5,6,7,8,9};
int[] y=x[:4];           // y={0,1,2,3}
int[] z=x[5:];           // z={5,6,7,8,9}
int[] w=x[:];            // w={0,1,2,3,4,5,6,7,8,9}, 与数组 x 相区别
```

如果 A 是一个非循环数组，则对任一指标使用负数值都是非法的。然而，如果指标超过数组长度，它们会被斯文地截短到此长度。

对于循环数组，切割  $A[m:n]$  仍然由集合  $[m, n)$  中的指标的单元构成，但现在允许负数值和超过数组长度的值。指标只不过是绕回。例如：

```
int[] x={0,1,2,3,4,5,6,7,8,9};
x.cyclic(true);
int[] y=x[8:15]; // y={8,9,0,1,2,3,4}.
int[] z=x[-5:5]; // z={5,6,7,8,9,0,1,2,3,4}
int[] w=x[-3:17]; // w={7,8,9,0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6}
```

注意对循环数组，有可能在一个切割中多次包含原数组中相同元素。不论原数组如何，由切割产生的数组总是非循环的。

如果一个切割的左指标和右指标是相同的，则结果是一个空白数组。如果被切割的数组是空白的，则结果是空白数组。任何左指标比右指标大的切割都会导致一个错误。

切割也可以被赋值，改变原数组的值。如果用来给切割赋值的数组长度与切割本身的长度不同，元素会被插入或从（原）数组中删除以适应（用来赋值的）数组。举例来说：

```
string[] toppings={"mayo", "salt", "ham", "lettuce"};
toppings[0:2]=new string[] {"mustard", "pepper"};
// 现在 toppings={"mustard", "pepper", "ham", "lettuce"}
toppings[2:3]=new string[] {"turkey", "bacon" };
// 现在 toppings={"mustard", "pepper", "turkey", "bacon", "lettuce"}
toppings[0:3]=new string[] {"tomato"};
// 现在 toppings={"tomato", "bacon", "lettuce"}
```

如果将一个数组赋给它自身的切割，则原数组的副本会被赋给此切割。亦即，像  $x[m:n]=x$  的代码等价于  $x[m:n]=\text{copy}(x)$ 。可以使用简化形式  $x[m:m]=y$  用来在数组  $x$  中恰在  $x[m]$  之前的位置处插入数组  $y$  的内容。

对循环数组，如果一个切割指向的单元到达数组的末尾然后又继续指向数组开头的单元，则此切割是桥接的。例如，如果  $A$  是一个长度为 10 的循环数组， $A[8:12]$ ,  $A[-3:1]$  和  $A[5:25]$  是桥接切割，而  $A[3:7]$ ,  $A[7:10]$ ,  $A[-3:0]$  与  $A[103:107]$  就不是。桥接切割只能在切割元素个数与待赋值元素个数严格相等时才能被赋值。否则的话，就没有一种明确的方法决定哪个新条目应该是  $A[0]$  从而报告错误。非桥接的切割可以用任意长度的数组赋值。

对循环数组  $A$ ，形如  $A[A.\text{length}:A.\text{length}]$  的表达式等价于表达式  $A[0:0]$ ，于是向此切割赋值将会在数组的开头将值插入。 $A.\text{append}()$  可用于向数组的末尾插入值。

向有任何重复单元的循环数组的切割赋值都是非法的。

## 6.13 类型转换

Asymptote 隐式地转换 `int` 到 `real`，`int` 到 `pair`，`real` 到 `pair`，`pair` 到 `path`，`pair` 到 `guide`，`path` 到 `guide`，`guide` 到 `path`，`real` 到 `pen`，`pair[]` 到 `guide[]`，`pair[]` 到 `path[]`，`path` 到 `path[]` 以及 `guide` 到 `path[]`，加上在 `three.asy` 中定义的多种三维类型转换。在赋值时以及在由可

行的函数标识尝试匹配函数调用时，隐式类型转换会自动尝试。在函数标识中，通过单独声明参数 `explicit` 可以阻止隐式类型转换，比如说在下例中防止一个歧义的函数调用，函数将输出 0：

```
int f(pair a) {return 0;}
int f(explicit real x) {return 1;}

write(f(0));
```

其他转换，比如说 `real` 到 `int` 或者 `real` 到 `string`，需要显式的类型转换：

```
int i=(int) 2.5;
string s=(string) 2.5;

real[] a={2.5,-3.5};
int[] b=(int []) a;
write(stdout,b); // 输出 2,-3
```

使用 `operator cast` 同样可以向用户自定义类型转换：

```
struct rpair {
    real radius;
    real angle;
}

pair operator cast(rpair x) {
    return (x.radius*cos(x.angle),x.radius*sin(x.angle));
}

rpair x;
x.radius=1;
x.angle=pi/6;

write(x);          // 输出 (0.866025403784439,0.5)
```

在定义新的类型转换操作时必须要小心。假设在一些代码中需要把所有的整数表示为 100 的倍数。要把它们转换为实数，应该先要把它们乘以 100。然而，直接的实现

```
real operator cast(int x) {return x*100;}
```

等价于一个无穷递归，因为结果 `x*100` 本身需要从整数转换为实数。作为替代，我们要用从 `int` 到 `real` 的标准转换：

```
real convert(int x) {return x*100;}
real operator cast(int x)=convert;
```

显式类型转换用 `operator ecast` 类似地实现。

## 6.14 导入

尽管 Asymptote 默认提供了许多特征，但一些应用仍需要外部 Asymptote 模块包含的特殊特征。例如，这几行

```
access graph;
graph.axes();
```

在二维图表中绘制  $x$  与  $y$  坐标轴。这里，指令在诸模块的一个全局字典中查找名为 `graph` 的模块，并将其置入一个名为 `graph` 的新变量。模块就是一个结构体，我们可以像通常在一个结构体中一样引用它的域。

经常，我们想要不必指定模块名就使用模块的函数。代码

```
from graph access axes;
```

把 `graph` 的 `axes` 域加进本地名字空间，从而，就可以写 `axes()`。如果给定的名字是重载的，所有这个名字的类型和变量都会加入。要加入多于一个名字，只要使用逗号分隔的列表：

```
from graph access axes, xaxis, yaxis;
```

通配符可以用来将模块中所有非私有（non-private）的域和类型加入本地名字空间：

```
from graph access *;
```

类似地，可以使用关键字 `unravel` 将一个结构体的非私有的域和类型加入本地名字空间：

```
struct matrix {
    real a,b,c,d;
}

real det(matrix m) {
    unravel m;
    return a*d-b*c;
}
```

命令

```
import graph;
```

是下面命令的一个方便的缩写形式：

```
access graph;
unravel graph;
```

即，`import graph` 首先将一个模块载入一个叫做 `graph` 的结构体，然后将其非私有的域和类型加入本地名字空间。这样的话，如果一个成员变量（或函数）被一个本地变量（或相同标识的函数）覆盖，原有的仍然可以通过用模块名限定来访问。

通配符导入在大多数情况下都工作良好，但通常并不知道一个模块的所有内部类型和变量，它们也可能由于模块作者增加或改变了模块的特征而改变。照此说来，在 `Asymptote` 文件开头加上 `import` 命令应审慎，以便导入的名字不会遮蔽其他导入的名字，这取决于它们引入的次序，并且如果引入的函数与后来定义的本地函数同名，还可能造成重载解析的问题。

要在将模块或域加入本地环境时对其重命名，使用 `as`：

```
access graph as graph2d;
from graph access xaxis as xline, yaxis as yline;
```

命令

```
import graph as graph2d;
```

是下面命令的一种方便的缩写形式：

```
access graph as graph2d;
unravel graph2d;
```

除了几个内部模块，如 `settings`，所有的模块都实现为 `Asymptote` 文件。当查找一个尚未加载的模块时，`Asymptote` 搜索标准路径（见 2.5 节搜索路径<sup>73</sup>）以匹配文件。对应此名字的文件被读入而其中的代码作为定义模块的结构体的体被解释。

如果文件名包含非字母数字的字符，用引号把它括起来：

```
access "/usr/local/share/asymptote/graph.asy" as graph;
from "/usr/local/share/asymptote/graph.asy" access axes;
import "/usr/local/share/asymptote/graph.asy" as graph;
```

模块导入自身（或互相循环导入）是错误。导入的模块名必须在编译时已知。

然而，你可以这样在运行时导入一个由字符串 `s` 确定的 `Asymptote` 模块：

```
eval("import "+s,true);
```

要有条件地运行一个 `asy` 文件数组，用

```
void asy(string format, bool overwrite ... string[] s);
```

使用输出文件格式 `format`，只有在 `overwrite` 为 `true` 或输出文件不存在时，文件才会被处理。可以求值一个包含在字符串 `s` 中的 `Asymptote` 表达式（然而没有任何返回值），使用：

```
void eval(string s, bool embedded=false);
```

<sup>73</sup> 此节译本不包括此节。——译者注

不需要在字符串末尾以分号结束。如果 `embedded` 为 `true`，字符串会在当前环境的顶层求值。如果 `embedded` 为 `false`（默认值），字符串会在一个不确定的环境求值，共享相同的 `settings` 模块（见第八章相关内容<sup>74</sup>）。

可以使用下列命令求值任意 Asymptote 代码（可以包含非转义的引号）：

```
void eval(code s, bool embedded=false);
```

这里 `code` 是一种特殊类型，像这样以 `quote {}` 包装起 Asymptote 代码来使用：

```
real a=1;
code s=quote {
    write(a);
};
eval(s,true);           // 输出 1
```

要逐字地包含一个文件 `graph` 的内容（就好像文件是内容被插入到那个点中），使用下列形式中的一种：

```
include graph;
include "/usr/local/share/asymptote/graph.asy";
```

要列出在一个以字符串 `s` 的内容为名字的模块中定义的全部全局函数和变量，用函数

```
void list(string s, bool imports=false);
```

如果 `imports` 为 `true`，导入的全局函数和变量也会被列出。

## 6.15 静态（static）

静态限定符将变量的内存地址分配在一个较高的外围环境。

对于一个函数体，变量在函数定义的块中分配；于是在代码

```
struct s {
    int count() {
        static int c=0;
        ++c;
        return c;
    }
}
```

中对每个 `s` 的对象都只有一个变量 `c` 的实例（而非每次调用 `count` 就有一个实例）。

类似地，在

---

<sup>74</sup>此节译本不包括此节。——译者注



```
int factorial(int n) {
    int helper(int k) {
        static int x=1;
        x *= k;
        return k == 1 ? x : helper(k-1);
    }
    return helper(n);
}
```

中对 `factorial` 的每次调用（而非对 `helper` 的每次调用）有一个 `x` 的实例，因而这是一个正确，然而丑陋的，阶乘的实现。

类似地，一个在结构体中声明的静态变量会在结构体定义的块中分配。所以，

```
struct A {
    struct B {
        static pair z;
    }
}
```

对创建每个类型 `A` 的对象创建一个对象 `z`。

在这个例子中，

```
int pow(int n, int k) {
    struct A {
        static int x=1;
        void helper() {
            x *= n;
        }
    }
    for(int i=0; i < k; ++i) {
        A a;
        a.helper();
    }
    return A.x;
}
```

对 `pow` 的每次调用有一个 `x` 的实例，因此这是指数函数的一个丑陋的实现。

循环结构在每次迭代时分配一个新的框架。这就是为什么高阶函数可以引用循环中特定迭代的变量：

```
void f();
for(int i=0; i < 10; ++i) {
```

```

int x=i;
if(x==5) {
    f=new void () { write(x); }
}
}
f();

```

这里，循环的每次迭代都有自己的变量 `x`，所以 `f()` 将写入 5。如果循环中的变量声明为静态的，它将在外围函数或结构体定义的地方分配（就好像它是在循环外面声明为 `static`）。例如，在

```

void f() {
    static int x;
    for(int i=0; i < 10; ++i) {
        static int y;
    }
}

```

中，`x` 与 `y` 二者将会在同一地方分配，这也正是 `f` 分配的地方。

语句也可以声明为静态的，此时它们会在外围函数或结构体定义的地方执行。不在函数或结构体定义内部包围的声明或语句已经在顶层了，因而静态限定符就失去意义了。这种情况下会给出一个警告。

鉴于结构体可以有静态域，对一个限定的名字，这个限定符是一个变量还是一个类型就并不总是清晰的了。例如，在

```

struct A {
    static int x;
}
pair A;

int y=A.x;

```

中，`A.x` 中的 `A` 是指结构体还是指复数（序对）变量呢？在 `Asymptote` 中的惯例是，如果同名的限定符中有一个非函数的变量，这个限定符指那个变量，而非类型。这与那个变量事实上究竟有什么域无关。

# 索引

!, 63, 64  
", 23  
\$, 64  
\$\$, 64  
%, 62, 64  
%=, 63  
&, 18, 63, 64  
&&, 63  
' , 23  
\*, 36, 46, 62, 64  
\*\*, 62, 64  
\*=, 63  
+, 36, 62, 64  
++, 63, 64  
+=, 63  
-, 62, 64  
--, 4, 63, 64  
---, 18, 64  
-=, 63  
-V, 1  
-glo, 54–56  
-nosafe, 54–56  
.., 4, 64  
/, 62, 64  
/\* \*/, 19  
//, 19  
/=: 63  
::, 18, 64  
<, 63, 64  
<<, 64  
<=: 63, 64  
==, 58, 63, 64  
>, 63, 64  
>=: 63, 64  
>>, 64  
? :, 63  
@, 64  
@@, 64  
[], 72, 74  
^, 46, 62–64  
^=: 63  
^^, 4, 5, 64  
0, 57  
  
[abort](#), 26  
[accel](#), 30  
[access](#), 82  
[acos](#), 70  
[acosh](#), 70  
[add](#), 42, 47, 51  
[alias](#), 58, 75  
[Align](#), 12  
[ailgn](#), 12  
[align](#), 14, 47  
[Allow](#), 45  
[AND](#), 63  
[angle](#), 21  
[animate](#), 56  
[append](#), 73  
[Arc](#), 28  
[arc](#), 27  
[ArcArrow](#), 7

ArcArrows, 7  
arclength, 30  
arctime, 30  
array, 74  
Arrow, 7  
arrow, 14  
arrowhead, 7  
arrowlength, 14  
Arrows, 7  
arrowsize, 7  
as, 83  
asin, 70  
asinh, 70  
Aspect, 48  
asy, 83  
asycolors.sty, 38  
atan, 70  
atan2, 70  
atanh, 70  
attach, 53  
AvantGarde, 41  
axialshade, 10  
azimuth, 22  
  
Bars, 7  
barsize, 7  
basealign, 40  
baseline, 14  
bbox, 49  
beep, 27  
BeginArcArrow, 7  
BeginArrow, 7  
BeginBar, 7  
BeginDotMargin, 8  
BeginMargin, 8  
BeginPenMargin, 8  
BeginPoint, 13  
beveljoin, 40  
  
binput, 55  
Black, 37  
black, 37  
Blank, 7  
blue, 37  
Bookman, 41  
bool, 20  
bool3, 20  
boutput, 55  
box, 15, 47  
bp, 2  
break, 19  
brick, 42  
brown, 37  
buildcycle, 32  
Bézier 曲线, 17  
    的几何性质, 17  
    离散生成, 17  
  
calculateTransform, 49  
cast, 81  
cbrt, 70  
CCW, 28  
cd, 54  
Center, 14  
change.system, 61  
change.user, 61  
chartreuse, 37  
checker, 42  
child.system, 61  
child.user, 61  
Circle, 27  
circle, 27  
clear, 55  
clip, 12  
close, 55  
cm, 2  
cmyk, 37

CMYK 颜色, 37  
code, 57, 84  
colatitude, 22  
colorless, 37  
colors, 37  
colorspace, 37  
comma, 55  
complement, 74  
concat, 75  
conf, 20  
continue, 19  
controls, 18  
controlSpecifier, 35  
convert, 56  
copy, 75  
cos, 70  
cosh, 70  
Courier, 41  
cputime, 61  
cross, 23  
crosshatch, 43  
CSV, 54  
csv, 78  
cubicroots, 78  
curl, 18  
curlSpecifier, 36  
currentpen, 36, 53  
currentpicture, 53  
currentprojection, 53  
CW, 28  
Cyan, 37  
cyan, 37  
cycle, 3, 4  
cyclic, 29, 35, 72  
cyclicflag, 72  
darkblue, 37  
darkbrown, 37  
darkcyan, 37  
darkgray, 37  
darkgreen, 37  
darkmagenta, 37  
darkolive, 37  
dashdotted, 39  
dashed, 39  
deepblue, 37  
deepcyan, 37  
deepgray, 37  
deepgreen, 37  
deepmagenta, 37  
Default, 7  
default, 20  
defaultformat, 9  
defaultpen, 36, 37  
degrees, 21  
delete, 56, 73  
diagonal, 77  
dimension, 78  
dir, 21, 22, 29, 30  
dirSpecifier, 35  
dirttime, 30  
do, 19  
dot, 9, 21, 23, 77  
dotfactor, 8  
DotMargin, 8  
DotMargins, 8  
Dotted, 39  
dotted, 39  
down, 3  
Draw, 7, 50  
draw, 7, 9, 11, 15  
E, 3  
ecast, 82  
ellipse, 15, 28, 47  
else, 19

empty, 47  
EndArcArrow, 7  
EndArrow, 7  
EndBar, 7  
EndDotMargin, 8  
endl, 55  
EndMargin, 8  
EndPenMargin, 8  
EndPoint, 13  
ENE, 3  
envelope, 15, 47  
eof, 55  
eol, 55  
erase, 1, 24, 47, 53  
erf, 70  
erfc, 70  
error, 54, 55  
eval, 83, 84  
evenodd, 4, 40  
exit, 26  
exp, 70  
expi, 21, 22  
explicit, 80  
expm1, 70  
extendcap, 39  
extension, 32  
  
fabs, 70  
false, 20, 57  
fft, 76  
file, 54  
Fill, 7, 50  
fill, 9  
FillDraw, 7, 50  
filldraw, 9  
filloutside, 10  
fillrule, 40  
filltype, 49  
  
find, 24, 74  
firstcut, 32  
fit, 49  
fixedscaling, 48  
flush, 55  
fmod, 70  
font, 41  
fontcommand, 41  
fontsize, 41  
fontsize.asy, 41  
for, 19  
format, 25  
frame, 47  
from, 82  
fuchsia, 37  
functionsshade, 12  
  
gamma, 70  
getc, 54  
getint, 56  
getpair, 56  
getreal, 56  
getstring, 56  
gettriple, 56  
gouraudshade, 11  
graphic, 14  
gray, 36, 37  
green, 37  
guide, 33  
  
hatch, 43  
heavyblue, 37  
heavycyan, 37  
heavygray, 37  
heavygreen, 37  
heavymagenta, 37  
heavyred, 37  
Helvetica, 41  
hex, 25

history, 56  
HookHead, 7  
hsv, 38  
hypot, 70  
  
I, 20  
identity, 46, 70, 77  
if, 19  
IgnoreAspect, 48  
import, 82  
inch, 2  
inches, 2  
include, 84  
init, 57, 61  
initialized, 73  
input, 54  
insert, 24, 73  
inside, 33  
int, 20  
interior, 33  
interp, 63  
intersect, 31  
intersectionpoint, 32  
intersectionpoints, 32  
intersections, 31  
intMax, 20  
intMin, 20  
inverse, 46, 78  
invisible, 37  
  
Jn, 70  
  
keys, 73  
  
Label, 13  
label, 3, 12, 13  
labelmargin, 8, 12  
Landscape, 49  
lastcut, 32  
L<sup>A</sup>T<sub>E</sub>X NFSS 字体, 41  
  
latitude, 22  
latticeshade, 10  
layer, 7  
left, 3  
LeftSide, 14  
legend, 8  
length, 20, 22, 24, 29, 35, 72  
libm 过程, 70  
lightblue, 37  
lightcyan, 37  
lightgray, 37  
lightgreen, 37  
lightmagenta, 37  
lightolive, 37  
lightred, 37  
lightyellow, 37  
line, 78  
linecap, 39  
linejoin, 40  
linetype, 39  
linewidth, 39  
list, 84  
locale, 25  
log, 70  
log10, 70  
log1p, 70  
longdashdotted, 39  
longdashed, 39  
longitude, 22  
  
Magenta, 37  
magenta, 37  
makepen, 44  
map, 74  
Margin, 8  
Margins, 8  
max, 33, 47, 51, 76  
maxbound, 21, 23

maxtimes, 32  
mediumblue, 37  
mediumcyan, 37  
mediumgray, 37  
mediumgreen, 37  
mediummagenta, 37  
mediumred, 37  
mediumpyellow, 37  
MidArcArrow, 7  
MidArrow, 7  
MidPoint, 13  
midpoint, 31  
min, 33, 47, 51, 76  
minbound, 21, 23  
minipage, 15  
mintimes, 32  
miterjoin, 40  
miterlimit, 40  
mm, 2  
Move, 45  
MoveQuiet, 45  
  
N, 3  
NE, 3  
new, 58, 74  
NewCenturySchoolBook, 41  
newframe, 47  
newl, 55  
nib, 44  
NoAlign, 12  
nobasealign, 40  
NoFill, 7, 50  
NoMargin, 8  
None, 7  
none, 55  
NOT, 63  
null, 55, 57, 58, 72  
nullpath, 33, 57  
0, 57  
object, 15  
olive, 37  
opacity, 41  
operator, 64  
operator cast, 81  
operator ecast, 82  
operator init, 57, 61  
OR, 63  
orange, 37  
orientation, 49  
outprefix, 49  
output, 54  
overwrite, 15, 45  
  
pack, 15  
pair, 2, 20  
pairs, 76  
Palatino, 41  
paleblue, 37  
palecyan, 37  
palegray, 37  
palegreen, 37  
palered, 37  
paleyellow, 37  
parent.system, 61  
parent.user, 61  
path, 27  
pattern, 42  
pen, 36  
PenMargin, 8  
PenMargins, 8  
picture, 48  
piecewisestraight, 29  
pink, 37  
point, 29, 35, 51  
polar, 22  
pop, 73



Portrait, 49  
position, 13  
postcontrol, 30  
postscript, 53  
PostScript 字体, 41  
pow10, 70  
precision, 55  
precontrol, 30  
prepend, 47  
private, 58  
pt, 2  
public, 58  
purple, 37  
push, 73  
  
quadraticroots, 78  
quit, 1  
quote, 84  
quotient, 62  
  
RadialShade, 50  
radialshade, 10  
radius, 30  
read1, 79  
read2, 79  
read3, 79  
readline, 56  
real, 20  
realDigits, 20  
realEpsilon, 20  
realMax, 20  
realMin, 20  
realmult, 21, 23  
red, 37  
reflect, 47  
Relative, 13, 14  
relpoint, 31  
reltime, 30  
remainder, 70  
  
rename, 56  
replace, 24  
restore, 53  
restricted, 58  
reverse, 24, 31, 35, 74  
rfind, 24  
rgb, 37  
RGB 颜色, 37  
right, 3  
RightSide, 14  
Rotate, 13  
rotate, 47  
roundbox, 15, 47  
roundcap, 39  
roundjoin, 40  
royalblue, 37  
  
safe, 57  
save, 53  
saveline, 56  
Scale, 13  
scale, 46, 49  
scroll, 56  
search, 75  
seconds, 26  
seek, 55  
seekeof, 55  
sequence, 74  
settings.scroll, 56  
Shift, 13  
shift, 46  
shiftless, 47  
shipout, 49  
SimpleHead, 7  
sin, 70  
single, 55  
sinh, 70  
size, 2, 29, 35, 48, 51

Slant, 13  
slant, 46  
sleep, 27  
slice, 32  
solid, 39  
solve, 77  
sort, 75  
split, 25  
springgreen, 37  
sqrt, 70  
squarecap, 39  
static, 84  
stdin, 55  
stdout, 55  
straight, 29  
string, 23, 25  
strokepath, 33  
struct, 58  
subpath, 31  
substr, 24  
sum, 76  
Suppress, 45  
SuppressQuiet, 45  
Symbol, 41  
system, 57  
  
tab, 55  
tan, 70  
tanh, 70  
tell, 55  
tension, 18  
tension at least, 18  
tensionSpecifier, 35  
tensorshade, 11  
tex, 53  
TEX 字体, 41  
texcolors, 38  
TeXHead, 7  
texpath, 15  
texpreamble, 54  
texreset, 54  
this, 58  
tile, 42  
time, 25, 26  
times, 31, 32  
TimesRoman, 41  
transform, 46  
transpose, 75, 76  
tridiagonal, 77  
triple, 21  
true, 20  
TrueMargin, 8  
truepoint, 51  
typedef, 27  
  
UnFill, 7, 50  
unfill, 12  
uniform, 74  
unit, 21, 22  
unitcircle, 4, 27  
unitsize, 3, 48  
unitsquare, 3  
unravel, 82  
up, 3  
UpsideDown, 49  
usepackage, 54  
usleep, 27  
  
void, 20  
  
while, 19  
white, 37  
windingnumber, 33  
write, 55, 61, 79  
  
x, 20, 21  
x 部分, 21, 22  
x11colors, 38

- XDR, 55
- `xinput`, 55
- XOR, 63
- `xoutput`, 55
- `xpart`, 21, 22
- `xscale`, 46
- y, 20, 21
- $y$  部分, 21, 22
- Yellow, 37
- yellow, 37
- Yn, 70
- `ypart`, 21, 22
- `yscale`, 46
- z, 21
- $z$  部分, 23
- ZapfChancery, 41
- ZapfDingbats, 41
- zerowinding, 4, 40
- `zpart`, 23
- 三元组, 21
- 三次样条曲线, 17
- 交互模式, 1
- 仿射变换, 46
- 余纬角, 22
- 入向切线, 17
- 出向切线, 17
- 函数, 65
  - 匿名函数, 66
  - 参数传值, 65
  - 参数传引用, 65
  - 递归, 67
  - 高阶函数, 66
- 切割 (slice), 79
- 初始式, 57
- 剪裁, 12
- 匿名函数, 66
- 单位向量, 21, 22
- 单引号, 23
- 卷曲参数, 18
- 历史记录, 56
- 叉积, 23
- 双引号, 23
- 反填充, 12
- 变换, 46
  - 倾斜, 46
  - 反射, 47
  - 复合, 46
  - 平移, 46
  - 恒同变换, 46
  - 放缩, 46
  - 旋转, 47
  - 逆, 46
  - 错切, 46
- 图, 48
- 图例, 8
- 圆, 27
- 圆弧, 27
- 基线, 14
- 填充, 9
- 声明变量, 19
- 复共轭, 20
- 复数, 20
- 大点, 2
- 字体命令, 41
- 字符串, 23
  - 拼接, 24
  - 空串, 24
- 实数, 20
- 实部, 20, 21
- 对齐, 12, 14
- 导入, 82
- 层, 7
- 布尔类型, 20
- 帧, 47

- 幅角, 21
- 引号, 23
- 引用, 58
- 张力, 18
- 循环, 19
- 恒同变换, 57
- 扩展布尔类型, 20
- 批处理模式, 1
- 按位运算, 63
- 控制流, 19
- 控制点, 17
- 控制结构, 19
- 描绘, 7
- 教程, 1
- 数据类型, 20
- 数组, 72
  - 切割 ( slice ) , 79
  - 空数组 ( null array ) , 72
  - 空白数组 ( empty array ) , 72
- 整数, 20
- 文件, 54
- 方向键, 1
- 映射, 23
- 条件测试, 19
- 构造函数, 60
- 标注, 12
- 标注路径, 13
- 标签, 3, 12
- 标签位置, 13
- 椭圆, 28
- 模长, 20, 22
- 注释, 19
- 浅复制, 72
- 深复制, 72
- 深比较, 58
- 渐变
  - Coons 渐变, 11
  - Gouraud 渐变, 11
  - 函数式渐变, 12
  - 填充连续一系列路径之间的区域, 11
  - 张量积渐变, 11
  - 放射梯度光滑渐变, 10
  - 格状梯度光滑渐变, 10
  - 轴向梯度光滑渐变, 10
- 灰阶颜色, 36
- 点, 2
- 点积, 21, 23
- 环形路径, 4
- 画点, 8
- 画笔, 36
  - 填充规则, 40
  - 字体, 41
  - 字体大小, 41
  - 尖角限量, 40
  - 数乘, 36
  - 文字对齐, 40
  - 相加, 36
  - 笔尖, 44
  - 线型, 39
  - 线宽, 39
  - 线端, 39
  - 覆写, 45
  - 连接样式, 40
  - 透明度, 41
  - 铺砌图案, 42
  - 颜色, 36
- 画路径上的点, 9
- 短杠, 7
- 空串, 57
- 空图, 57
- 空帧, 57
- 空引用, 58
- 空数组 ( null array ) , 72
- 空白数组 ( empty array ) , 72

空类型, 20  
箭头, 7  
类型转换, 61, 80  
纬度角, 22  
经度角, 22  
结构体, 58  
    引用, 58  
    空引用, 58  
结点, 17  
继承, 61  
罗盘方向, 3

自动补全, 1  
英寸, 2  
虚函数, 61  
虚部, 20, 21  
路向, 33  
路径, 27  
边距, 8  
运算符  
    前缀运算符, 63  
    用户自定义运算符, 63  
    算术和逻辑运算符, 62  
    自（赋值）运算符, 63  
    重载, 64

逆变换, 46  
逆时针, 28  
逐元素乘积, 21, 23  
遍历数组, 19  
隐式放缩, 64  
静态（static）, 84  
顺时针, 28  
颜色

    Black, 37  
    black, 37  
    blue, 37  
    brown, 37  
    chartreuse, 37

    Cyan, 37  
    cyan, 37  
    darkblue, 37  
    darkbrown, 37  
    darkcyan, 37  
    darkgray, 37  
    darkgreen, 37  
    darkmagenta, 37  
    darkolive, 37  
    deepblue, 37  
    deepcyan, 37  
    deepgray, 37  
    deepgreen, 37  
    deepmagenta, 37  
    fuchsia, 37  
    gray, 37  
    green, 37  
    heavyblue, 37  
    heavycyan, 37  
    heavygray, 37  
    heavygreen, 37  
    heavymagenta, 37  
    heavyred, 37  
    lightblue, 37  
    lightcyan, 37  
    lightgray, 37  
    lightgreen, 37  
    lightmagenta, 37  
    lightolive, 37  
    lightred, 37  
    lightyellow, 37  
    Magenta, 37  
    magenta, 37  
    mediumblue, 37  
    mediumcyan, 37  
    mediumgray, 37  
    mediumgreen, 37  
    mediummagenta, 37

mediumred, 37  
mediummyellow, 37  
olive, 37  
orange, 37  
paleblue, 37  
palecyan, 37  
palegray, 37  
palegreen, 37  
palered, 37  
paleyellow, 37  
pink, 37  
purple, 37  
red, 37  
royalblue, 37  
springgreen, 37  
white, 37  
Yellow, 37  
yellow, 37

颜色空间, 37

高阶函数, 66

默认画笔, 57